CERTIFICATION OF APPROVAL

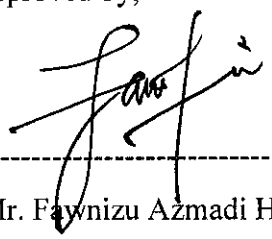
OPTIMIZATION OF TWOFISH ENCRYPTION ALGORITHM ON FPGA

by

ANANDA RAJA A/L DORE RAJA

A project dissertation submitted to the
Electrical & Electronics Engineering Programme
Universiti Teknologi PETRONAS
In partial fulfillment of the requirement for the
BACHELOR OF ENGINEERING (Hons)
(ELECTRICAL & ELECTRONICS ENGINEERING)

Approved by,



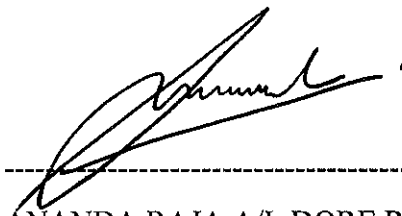
(Mr. Fawnizu Azmadi Hussin)
Lecturer, UTP.

Fawnizu Azmadi Hussin
Lecturer
Electrical & Electronics Engineering
New Academic Block NO 22
Universiti Teknologi PETRONAS
31750 Tronoh
Perak Darul Ridzuan, MALAYSIA.

UNIVERSITI TEKNOLOGI PETRONAS
TRONOH, PERAK
DECEMBER 2004

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own except as specified in the references and acknowledgements and that the original work contained herein have not been undertaken or done by unspecified sources or persons.



ANANDA RAJA A/L DORE RAJA

ABSTRACT

The demand for efficient and secure ciphers has given rise to a new generation of block ciphers capable of providing increased protection at lower cost. Among these new algorithms is Twofish. Twofish is a promising 128-bit block which was one of the 5 finalists in the National Institute of Standards and Technology organized competition as the Advanced Encryption Standard. The aim of the competition was to find a suitable candidate to replace DES at the core of many encryption systems worldwide.

Twofish can work with variable key lengths: 128, 192 or 256 bits. In this report, only a version of 128-bit key length was discussed. Twofish has 6 main building blocks; Feistel Networks, whitening, S-boxes, MDS Matrices, Pseudo Hadamard Transforms and Key Schedule. Twofish is a 16 round Feistel network with a bijective F function, which corresponds to 8 cycles. The whitening technique employed substantially increases the difficulty of keysearch attacks against the remainder of the cipher. Twofish uses 4 different, bijective, key-dependent, 8-by-8 bit S-boxes. Twofish uses a single 4 by 4 MDS matrix over GF (2^8).This is one of the 2 main diffusion elements of Twofish. There is also Reed-Solomon code with the MDS property used in the key schedule; this doesn't add diffusion to the cipher but does add diffusion to the key schedule.) Besides that, Twofish also uses a 32 bit Pseudo Hadamard Transform to mix the outputs from its 2 parallel 32-bit g functions. Finally, Twofish needs a lot of key material, and has complicated key schedule. To facilitate analysis, the key schedule uses the same primitives as the round function. Except for 2 additional rotations, each pair of expanded key words is constructed by applying the Twofish round function (with key-dependent).

For this project, 2 different optimized designs were implemented. The first design (Design 1) was implemented with minimum hardware resources usage, using a single F-Function (modified) and was optimized with reasonable latency, throughput and throughput per gate. As for the second design (Design 2) was implemented with reasonably minimum hardware resources using 4 units of F-Function(modified) of Design 1, minimum hardware resources usage, very small latency, very high throughput and very high throughput per gate. Furthermore, both Design 1 and Design 2 were implemented with zero keying and function as encryptor/decryptor. Both Design 1 and Design 2 were written using VHDL, simulated using ALDEC, synthesized using XILINX Synthesizing Tools, implemented using XILINX ISE6.2i implementation tools and download onto the Spartan 2 FPGA board using BEDLOAD utility program.

As a conclusion this Final Year Project is quite successful because all the objectives have been met successfully.

ACKNOWLEDGEMENT

My biggest thanks go to my Final Year Supervisor, Mr. Noohul Basheer who found time and energy in his busy schedule to guide me towards completing this project successfully.

I thank my co-supervisor, Mr. Fawnizu for giving me valuable advice to improve my design.

I also thank all UTP technicians who had lent their help to me while I was .completing this project.

Special thanks are due to the following individuals who had given valuable technical advice on hardware description language and hardware programming that helped me to complete my project successfully.

- Mr. Ettikan Karuppiah Kandasamy — INTEL, Penang.
- Mr. Avinansh Ravindranathan — INTEL, Penang.
- Miss Vincci Lee Hoong Ming — INTEL, Penang.
- Miss Teresa Kok — Insight Memec,Singapore.

Finally, I am grateful to UTP for providing necessary equipment and facilities that helped in the completion of my project.

TABLE OF CONTENTS

CERTIFICATION OF APPROVAL i

CERTIFICATION OF ORIGINALITY ii

ABSTRACT iii

ACKNOWLEDGEMENT iv

1. CHAPTER 1: INTRODUCTION 1
1.1 BACKGROUND STUDY 1
1.2 PROBLEM STATEMENT 1
1.3 OBJECTIVES 2
1.4 SCOPE OF STUDY 3

2. CHAPTER 2: LITERATURE REVIEW AND THEORY 4
2.1 INTRODUCTION 5
2.2 TWOFISH DESIGN GOALS 6
2.3 TWOFISH BUILDING BLOCKS 8
2.3.1 Feistel Networks 8
2.3.2 S-Boxes 9
2.3.3 MDS Matrices 9
2.3.4 Pseudo Hadamard Transform 10
2.3.5 Whitening 10
2.3.6 Key Schedule 11
2.4 TWOFISH ENCRYPTION ALGORITHM – A DETAIL DESCRIPTION 11
2.4.1 The Function F 13
2.4.2 The Function g 13
2.4.3 The Key Schedule 14
2.4.3.1 Additional Key Lengths 16
2.4.3.2 The Function h 16
2.4.3.3 Key-dependent S-boxes 17
2.4.3.4 The Expanded Key Words K_j 19
2.4.3.5 The Permutations q_0 and q_1 19
2.4.4 Round Function Overview 20

3. CHAPTER 3: METHODOLOGY 22
3.1 PROCEDURE IDENTIFICATION 22
3.2 TOOLS 22

4. CHAPTER 4: PROJECT WORK 23
4.1 DESIGN IMPLEMENTATION 23
4.1.1Design Decisions 24
4.1.1.1 Building Blocks 24

4.1.1.1.1 Q-Permutations 25
4.1.1.1.2 S-Boxes 26
4.1.1.1.3 Maximum Distance Separable Matrix 27
4.1.1.1.4 Reed-Solomon Matrix 28
4.1.1.1.5 Operation Selector 29
4.1.2 Integration and Overall Structure 30
4.1.2.1 Input Register Module 31
4.1.2.2 Outer Register Module 32
4.1.2.3 Key Register Module 33
4.1.2.4 Modified F-Function 35
4.1.2.5 Design Overall Structure 39
4.1.2.6 Controller 42
4.2 PROGRAMMING STRATEGY 46
4.2.1 TWOFISH CORE 47
4.2.2 FULL ADDER 49
4.2.3 CIPHERTEXT 49
4.2.4 CLEARTEXT/PLAINTEXT 53
4.2.5 CONTROLLER 54
4.2.6 KEYMODULE 66
4.2.7 MODIFIED_F 66
4.2.8 OPSELECT 73
4.2.9 32 BIT REGISTER 74
4.2.10 WRAPPER 75

5. CHAPTER 5: RESULTS & DISCUSSION 79
5.1 SIMULATION 79
5.2 TEST VECTOR 1– DESIGN 1 79
5.2.1Encryption 79
5.2.1.1 Load Key 79
5.2.1.2 Start Encryption 81
5.2.2 Decryption 83
5.2.2.1 Load Key 83
5.2.2.2 Start of Decryption 84
5.3 TEST VECTOR 2– DESIGN 1 86
5.3.1 Encryption 86
5.3.2 Decryption 87
5.4 TEST VECTOR 1– DESIGN 2 89
5.4.1Encryption 89
5.4.1.1 Load Key 89
5.4.1.2 Start Encryption 91
5.4.2 Decryption 93
5.4.2.1 Load Key 93
5.4.2.2 Start of Decryption 95
5.5 TEST VECTOR 2– DESIGN 97
5.5.1 Encryption 97
5.5.2 Decryption 98

5.6 DESIGN IMPLEMENTATION 100
5.7 GENERATE PROGRAMMING FILE & DOWNLOADING 102

6. **CHAPTER 6: DISCUSSION** 103
6.1 ORIGINAL DESIGN BY THE AUTHOR 103
6.2 DESIGN 1 105
6.3 DESIGN 2 107
6.4 PERFORMANCE DIFFERENCE BETWEEN DESIGN 1 AND DESIGN 2 ON SPARTAN 2-XC2S200-5PQ208 111
6.5 PERFORMANCE OF DESIGN 1 AND DESIGN 2 WHEN IMPLEMENTED ON SPARTAN 3 –XC3S400-5FG456 112
6.6 PERFORMANCE COMPARISON OF DESIGN 1 WHEN IMPLEMENTED ON SPARTAN 2 - XC2S200-5PQ208 & SPARTAN 3 –XC3S400-5FG456 113
6.7 PERFORMANCE COMPARISON OF DESIGN 2 WHEN IMPLEMENTED ON SPARTAN 2 - XC2S200-5PQ208 & SPARTAN 3 –XC3S400-5FG456 114
6.8 FACTORS CAUSING DIFFERENCES BETWEEN ONE IMPLEMENTATION AND ANOTHER IMPLEMENTATION OF FPGA 114
6.9 PERFORMANCE COMPARISON 116

7. **CHAPTER 7: CONCLUSION AND RECOMMENDATION** 118
7.1 CONCLUSION 118
7.2 RECOMMENDATION 119

8. **CHAPTER 8: REFERENCE** 120

9. **APPENDIX A: MDS MATRIX** 121

10. **APPENDIX B: REED SOLOMON MATRIX** 130

11. **APPENDIX C: SOURCE CODE FOR DESIGN 2** 220

LIST OF FIGURES

Figure 1: Schematic Diagram of Twofish Encryption Algorithm
Figure 2: Function h
Figure 3: A view of a Single Round F Function (128-bit Key)
Figure 4: Project Development Block
Figure 5: Q-Permutation
Figure 6: S-boxes
Figure 7: MDS
Figure 8: Reed-Solomon Matrix
Figure 9: Operation Selector
Figure 10: Building Block for Operation Selection
Figure 11: Input Register Module for Design 1
Figure 12: Output Register Module for Design 1
Figure 13: Output Register Module for Design 2
Figure 14: Key Register Module
Figure 15: Generalized F-Function of Design 1
Figure 16: Generalized F-Function of Design 2
Figure 17: Structure of the Cipher of Design 1
Figure 18: Structure of the Cipher of Design 2
Figure 19: The Controller of Design 1
Figure 20: The Controller of Design 2
Figure 21: Hierarchy of the Design Flow
Figure 22: Block Diagram of the Twofish Core – with I/O pins
Figure 23: Block Diagram of the Full Adder
Figure 24: Block Diagram of the Ciphertext of Design 1
Figure 25: Block Diagram of the Ciphertext of Design 2
Figure 26: Block Diagram of the Cleartext/Plaintext
Figure 27: Finite State Machine of the Design 1
Figure 28: Overall Design with some of the Control Signals for Design 1
Figure 29: Block Diagram of the Controller for Design 1
Figure 30: Finite State Machine of the Design 2
Figure 31: Overall Design with some of the Control Signals for Design 2
Figure 32: Block Diagram of the Controller for Design 2
Figure 33: Block Diagram of the Keymodule
Figure 34: Block Diagram of the Modified_F of Design 1
Figure 35: Block Diagram of Evenkeygenerator
Figure 36: Block Diagram of Oddkeygenerator
Figure 37: Block Diagram of Evenplaintextgenerator
Figure 38: Block Diagram of Oddplaintextgenerator
Figure 39: Block Diagram of the Opselect
Figure 40: Block Diagram of the 32-bit Register
Figure 41: Wrapper for Both Design 1 and Design 2
Figure 42: Idle – Text Vector 1

Figure 43: Load Key – Text Vector 1
Figure 44: Start Encryption – Text Vector 1
Figure 45: End of Encryption – Text Vector 1
Figure 46: Initialization of Decryption – Text Vector 1
Figure 47: Decryption – Loading Key – Text Vector 1
Figure 48: Start of Decryption – Text Vector
Figure 49: End of Decryption – Text Vector
Figure 50: Initialization of Input Values – Test Vector 2
Figure 51: End of Encryption – Test Vector 2
Figure 52: Initialization of Input Values – Test Vector 2
Figure 53: End of Decryption – Test Vector 2
Figure 54: Idle – Text Vector 1
Figure 55: Load Key – Text Vector 1
Figure 56: Start of Encryption – Text Vector 1
Figure 57: End of Encryption – Text Vector 1
Figure 58: Initialization of Decryption – Text Vector 1
Figure 59: Decryption – Loading Key – Text Vector 1
Figure 60: Start of Decryption – Text Vector
Figure 61: End of Decryption – Text Vector
Figure 62: Initialization of Input Values – Test Vector 2
Figure 63: End of Encryption – Test Vector 2
Figure 64: Initialization of Input Values – Test Vector 2
Figure 65: End of Decryption – Test Vector 2Figure 66: B3-SPARTAN2+ Board
Figure 67: A View of a Complete Single Round F –Function of the Original Design as Proposed by the Author
Figure 68: Generalized Modified F-Function
Figure 69: A View of a Complete Single Round F –Function of Design 2
Figure 70: Generalized F-Function of Design 2

LIST OF TABLES

Table 1: Twofish Core
Table 2: Full Adder
Table 3: Ciphertext – Design 1
Table 4: Ciphertext – Design 2
Table 5: Cleartext/Plaintext
Table 6: Controller – Design 1
Table 7: Generated Control Signals for the Corresponding States for Design 1
Table 8: Controller – Design 2
Table 9: Keymodule
Table 10: Modified_F – Design 1

Table 11: Evenkeygenerator
Table 12: Oddkeygenerator
Table 13: Evenplaintextgenerator
Table 14: Oddplaintextgenerator
Table 15: Opselect
Table 16: 32-bit Register
Table 17: RAM for Storing Data Blocks
Table 18: Wrapper
Table 19: Pin Assignment to the Corresponding I/O
Table 20: Generated Sequence of Values Based on Design 1
Table 21: Generated Sequence of Values Based on Design 1
Table 22: Output Performance of Design 1 on Spartan 2- XC2S200-5PQ208
Table 23: Generated Sequence of Values Based on Design 2
Table 24: Output Performance of Design 2 on Spartan 2- XC2S200-5PQ208
Table 25: Performance Difference of Design 1 and Design 2 on Spartan 2- XC2S200-5PQ208
Table 26: Performance Difference of Design 1 and Design 2 on Spartan 3 –XC3S400-5FG456
Table 27: Performance Difference of Design 1 on Spartan 2 - XC2S200-5PQ208 & Spartan 3 –XC3S400-5FG456
Table 28: Performance Difference of Design 2 on Spartan 2 - XC2S200-5PQ208 & Spartan 3 –XC3S400-5FG456
Table 29: Performance Comparison With Other Implementations

CHAPTER 1: INTRODUCTION

1.1 BACKGROUND STUDY

With the explosive growth in computer systems and their interconnections via networks has increased the dependence of both organizations and individuals on the information stored and communicated using these systems. However in these networking environments there are no such guarantees that all kinds of information (databases, video programs, telecommunications etc...) can avoid unauthorized access, because the transmission medium is open, which implies that anyone with the appropriate protocol analyzer can eavesdrop as well. This in turn has led to a heightened awareness of the need to protect data and resources from disclosure to guarantee the authencity of data and messages and to protect systems from network based attacks. Many of the cryptographic algorithms that have been developed are being used in software implementations on computers (e.g. to have protection of coded passwords for users). For low complexity type of applications, such as the protection of information in files and databases this is probably the most economic solution. However, a number of applications require such high throughputs for encryption/decryption process that they cannot be executed on a normal general purpose microprocessor. These applications require dedicated ASIC or FPGA implementations. In the past many VLSI implementations in block cipher have been proposed such as DES, IDEA, SAFER, etc. In this report, I present the implementation of Twofish encryption algorithm on FPGA. This architecture can be used to implement on high speed networking.

1.2 PROBLEM STATEMENT

With the large and growing number of Internet and wireless communication users has led to an increasing demand of security measures and devices for protecting the user data transmitted over the open channels. Two kinds of cryptographic systems can be used for that purpose i.e., the symmetric-key and asymmetric key crypto systems. The symmetric-key cryptography (such as DES, AES, Blowfish and Twofish) uses an identical key between the sender and receiver, both to encrypt the message text and decrypt the cipher text. The asymmetric-key cryptography (such as RSA), uses different keys for encryption and decryption, eliminating the key transporting

dilemma. Because of its high speed, the symmetric key cryptography is more suitable for the encryption of large amount of data. Almost the last 30 years, DES (Data Encryption Standard) has been the standard encryption algorithm around [1]. Despite the popularity, DES' key length is too short for acceptable commercial security. The 64-bit block length shared by DES and most well known ciphers opens it up to attacks when large amount of data are encrypted under the same key. NIST (National Institute of Standards and Technology, or NIST) specified that new generation block ciphers should have longer key length, larger block size, faster speed and greater flexibility. Twofish encryption algorithm was one of the finalists of AES. It is a 128 bit symmetric block cipher with variable key lengths of 128, 192 and 256 bits. Furthermore it has no weak keys. For this project, I plan to use Twofish encryption algorithm as the implementation algorithm.

Twofish has many qualities that make it interesting for this project. It has been designed with hardware implementation in mind and can thus be mapped efficiently to hardware devices such as FPGA and SmartCards. It also offer different possibilities of trade-offs between space and speed and can be pipelined. The estimated gate count would be 14000 and 80000. In this project we'll be looking at the implementation of the Twofish encryption algorithm on FPGA.

1.3 OBJECTIVES

The objectives of this project are as follows:

- To understand Twofish Encryption Algorithm.
- To understand the complex mathematical expression especially Galois Field and Permutation that makes up the algorithm.
- To understand how to translate Twofish Encryption Algorithm into a hardware module.
- To implement Twofish Encryption Algorithm on FPGA.
- To optimize the design of the algorithm onto the FPGA
- To use VHDL in coding the algorithm onto the FPGA
- To understand the nature of the FPGA

1.4 SCOPE OF STUDY

The scope of study basically covers the implementation of Twofish Encryption algorithm and the various building blocks namely Feistel Network, whitening, S-boxes, MDS Matrices, PHT and Key Schedule. This is followed by exploring and then implementing with an optimized design onto the FPGA. The design would be coded using VHDL.

CHAPTER 2: LITERATURE REVIEW AND THEORY

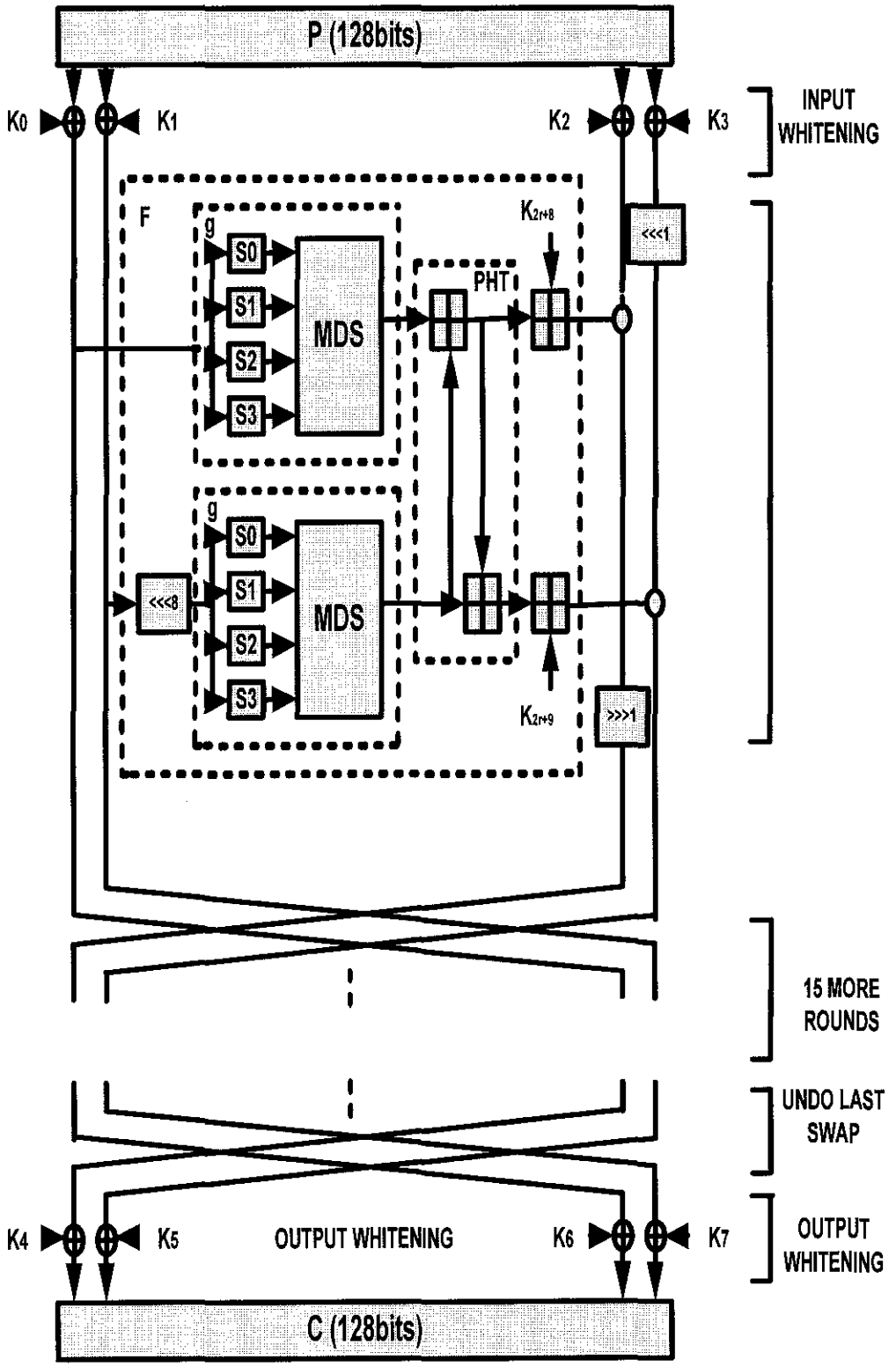


Figure 1: Schematic Diagram of Twofish Encryption Algorithm

2.1 INTRODUCTION

DES or Data Encryption Standard is one of the most widely used encryption algorithm in the world. It originated between 1972 and 1974, when the National Bureau of Standards issued a public request for an encryption standard. Even though it was a very successful and popular algorithm, it was plagued with controversy. This was because, many cryptanalysts did not welcome the development of DES whish was done in a closed door policy. They feared that NBS had embedded some backdoor to the algorithm. In short the development was not transparent. The debate about whether DES' key is too short for acceptable commercial security has been going on for many years. Recent advances in distributed key search techniques have left no doubt in anyone's mind that its key is simply too short for today's security applications especially with the advent of powerful computers. Triple- DES which is basically the looping of DES by 3 times has emerged as an interim solution in many high-security applications, such as banking. The disadvantage of Triple-DES is it is slow and very time consuming. Some of the inherent problem with DES is the it could be broken if the data has been encrypted many times using the same key. Besides that in 1990, a powerful machine was developed that brute forcedly broke DES. With the growing request from the academic and industrial world to replaced DES, the National Institute of Standards and Technology (NIST) announced a new program called the Advanced or American Encryption Standard program in 1997. Comments and feedback from the public were collected and gathered on the proposed standard before the NIST eventually issued a call for algorithms to satisfy the standard. The intention is for NIST to make all submissions public and eventually, through a process of public review and comment, choose a new encryption standard to replace DES. NIST's call requested a block cipher. Block ciphers can be used to design stream ciphers with a variety of synchronization and error extension properties, one-way hash functions, message authentication codes, and pseudo-random number generators. Because of this exibility, they are the workhorse of modern cryptography. NIST specified several other design criteria: a longer key length, larger block size, faster speed, and greater exibility. While no single algorithm can be optimized for all needs, NIST intends AES to become the standard symmetric algorithm of the next decade.

Twofish is an encryption algorithm that was jointly created by Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall and Niels Ferguson which was a submission to the AES selection process. It meets the entire required NIST criteria 128-bit block; 128-, 192-, and 256-bit key; efficient on various platforms; etc. and some strenuous design requirements, performance as well as cryptographic. Twofish can:

- Encrypt data at 285 clock cycles per block on a Pentium Pro, after a 12700 clock-cycle key setup.
- Encrypt data at 860 clock cycles per block on a Pentium Pro, after a 1250 clock-cycle key setup.
- Encrypt data at 26500 clock cycles per block on a 6805 smart card, after a 1750 clock-cycle key setup.

2.2 TWOFISH DESIGN GOALS

Twofish encryption algorithm was designed carefully to meet all the criteria set by NIST [2, 9]. Among the major design criteria specified by NIST are as follows:

- A 128-bit symmetric block cipher.
- Key lengths of 128 bits, 192 bits, and 256 bits.
- No weak keys.
- Highly efficient on both the Intel Pentium Pro and other software and hardware platforms.
- Flexible design: e.g., accept additional key lengths; be implementable on a wide variety of platforms and applications; and be suitable for a stream cipher, hash function, and MAC.
- Simple design, both to facilitate ease of analysis and ease of implementation.

Additionally, the authors imposed the following performance criteria on their design namely [2]:

- Accept any key length up to 256 bits.

- Encrypt data in less than 500 clock cycles per block on an Intel Pentium, Pentium Pro, and Pentium II, for a fully optimized version of the algorithm.
- Shall be capable of setting up a 128-bit key (for optimal encryption speed) in less than the time required to encrypt 32 blocks on a Pentium, Pentium Pro, and Pentium II.
- Encrypt data in less than 5000 clock cycles per block on a Pentium, Pentium Pro, and Pentium II with no key setup time.
- Shall not contain any operations that make it inefficient on other 32-bit microprocessors.
- Shall not contain any operations that make it inefficient on 8-bit and 16-bit microprocessors.
- Shall not contain any operations that reduce its efficiency on proposed 64-bit microprocessors.
- Shall not include any elements that make it inefficient in hardware.
- Have a variety of performance tradeoffs with respect to the key schedule.
- Encrypt data in less than less than 10 milliseconds on a commodity 8-bit microprocessor.
- Shall be implementable on an 8-bit microprocessor with only 64 bytes of RAM.
- Be implementable in hardware using less than 20,000 gates.

Besides that the authors' cryptographic goals were as follows:

- 16-round Twofish (without whitening) should have no chosen-plaintext attack requiring fewer than 280 chosen plaintexts and less than 2^N time, where N is the key length.
- 12-round Twofish (without whitening) should have no related-key attack requiring fewer than 264 chosen plaintexts, and less than $2^{N/2}$ time, where N is the key length.

Finally, the authors imposed the following flexibility goals:

- Should have variants with a variable number of rounds.

- Should have a key schedule that can be precomputed for maximum speed, or computed on-the fly for maximum agility and minimum memory requirements. Additionally, it should be suitable for dedicated hardware applications: e.g., no large tables.
- Shall be suitable as a stream cipher, one-way hash function, MAC, and pseudo-random number generator, using well-understood construction methods.
- Shall be a family-key variant to allow for different, non-interoperable, versions of the cipher.

2.3 TWOFISH BUILDING BLOCKS

2.3.1Feistel Networks

A *Feistel network* is defined to be a general method of transforming any function (usually called the *F* function) into a permutation. Feistel network was first invented by Horst Feistel. The method was first used in an encryption algorithm developed by himself called Lucifer. It gained popularity and was further popularized when it was used in DES. Among the major algorithms that uses Feistel Network are as follows [1]: -

- Blowfish
- LOKI
- GOST
- RC5

The fundamental building block of a Feistel network is the *F* function: a key-dependent mapping of an input string onto an output string. A *F* function is always non-linear and possibly non-surjective:

$$F: \{0, 1\}^{n/2} \times \{0, 1\}^{N} \rightarrow \{0, 1\}^{n/2}$$

where n is the block size of the Feistel Network, and F is a function taking n=2 bits of the block and N bits of a key as input, and producing an output of length n=2 bits. In each round, the "source block" is the input to F, and the output of F is xored with the target block," after which these two blocks swap

places for the next round. The important observation that can be made here is that a F- function which may be a very weak encryption algorithm when fed into the Feistel network and repeatedly iterated, the function becomes stronger and stronger. This is one method of increasing the immunity of an algorithm from potential attacks. In Feistel Network, a cycle consists of 2 rounds. In one cycle, every bit of the text block has been modified once. Twofish is a 16-round Feistel network with a bijective *F* function which corresponds to 8 cycles.

2.3.2 S - Boxes

Twofish uses key dependent S-boxes. S-box is defined to be a table-driven non-linear substitution operation used in most block ciphers. Fixed S-boxes (e.g. DES) allow attackers to study S-boxes and find weak key points but with key dependent S-boxes, attacker doesn't know what the S-boxes are. It is basically defense for an "unknown attack". S-boxes vary in both input size and output size, and can be created either randomly or algorithmically. S-boxes were first used in Lucifer, then DES, and afterwards in most encryption algorithms. Twofish uses four different, bijective, key-dependent, 8-by-8-bit S-boxes. These S-boxes are built using two fixed 8-by-8-bit permutations and key material. These S-boxes can either be precomputed for a specific key or computed on the fly for every required value. This provides a lot of flexibility and tradeoffs both in hardware and software.

2.3.3 MDS Matrices

In pure mathematics, maximum distance separable (MDS) code over a field is defined to be a linear mapping from *a fi*eld elements to *b* field elements, producing a composite vector of *a*+*b* elements, with the property that the minimum number of non-zero elements in any non-zero vector is at least *b* + 1. Put another way, the distance" (i.e., the number of elements that differ) between any two distinct vectors produced by the MDS mapping is at least *b* + 1. It can easily be shown that no mapping can have a larger minimum distance between two distinct vectors, hence the term maximum distance separable. MDS mappings can be represented by an MDS matrix consisting of *a*+ *b* elements. Reed-Solomon (RS) error-correcting codes are known to be MDS. A

necessary and sufficient condition for an $a+ b$ matrix to be MDS is that all possible square submatrices, obtained by discarding rows or columns, are non-singular Twofish uses a single 4-by-4 MDS matrix over GF (2^8). **This is one of the main diffusion elements of Twofish.** (There is also an RS-code with MDS property used in the key schedule; this doesn't add diffusion to the cipher, but does add diffusion to the key schedule).

2.3.4 Pseudo-Hadamard Transforms

A pseudo-Hadamard transform (PHT) is very important in encryption algorithm. PHT is defined to be a simple mixing operation that runs quickly in software. Given two inputs, a and b, the 32-bit PHT is defined as:

$$a' = a + b \bmod 2^{32}$$
$$b' = a + 2b \bmod 2^{32}$$

Twofish uses a 32-bit PHT to mix the outputs from its two parallel 32-bit g functions. This PHT can be executed in two opcodes on most modern microprocessors, including the Pentium family. **This is the second main diffusion element in Twofish.**

2.3.5 Whitening

In the world of cryptography, whitening is very important. Whitening is defined to be the technique of xoring key material before the first round and after the last round. It was proven by many cryptanalysts that whitening substantially increases the difficulty of key search attacks against the remainder of the cipher. In the authors' attacks on reduced-round Twofish variants, the authors discovered that whitening substantially increased the difficulty of attacking the cipher, by hiding from an attacker the specific inputs to the first and last rounds' F functions. Twofish xors 128 bits of subkey before the first Feistel round and another 128 bits after the last Feistel round. These subkeys are calculated in the same manner as the round subkeys, but are not used anywhere else in the cipher.

2.3.6 Key Schedule

Another important building block of Twofish is the Key Schedule. The key schedule is the means by which the key bits are turned into round keys that the cipher can use. The requirements call for a variable length key that is whether we use 128, 196 or 256 bits of key length. The easiest way of using this is to have a key schedule that expands a variable length key to a fixed set of expanded key values. Twofish needs a lot of key material, and has a complicated key schedule. To facilitate analysis, the key schedule uses the same primitives as the round function. Except for 2 additional rotations, each pair of expanded key words is constructed by applying the Twofish round function (with key-dependent S-boxes) to a fixed input.

2.4 TWOFISH ENCRYPTION ALGORITHM – A DETAIL EXPLANTION

Twofish encryption algorithm is one of the leading encryption algorithms in the world. It was designed with many considerations in mind. The authors have great experience in the world of cryptography and as a result developed a very strong algorithm that meets all NIST's requirements. Figure 1 above shows an overview of the Twofish block cipher. Twofish uses a 16-round Feistel-like structure with additional whitening of the input and output. The only non-Feistel elements are the 1-bit rotates. The rotations can be moved into the F function to create a pure Feistel structure, but this requires an additional rotation of the words just before the output whitening step. Besides that, the plaintext is split into four 32-bit words. In the input whitening step, these are xored with four key words. This is followed by sixteen rounds. In each round, the two words on the left are used as input to the g functions. (One of them is rotated by 8 bits first.) The g function consists of four byte-wide key-dependent S-boxes, followed by a linear mixing step based on an MDS matrix. Next, the results of the two g functions are combined using a Pseudo- Hadamard Transform (PHT), and two keywords are added. These two results are then xored into the words on the right (one of which is rotated left by 1 bit first, the other is rotated right afterwards). The left and right halves are then swapped for the next round. After all the rounds, the swap of the last round is reversed, and the four words are xored with

four more key words to produce the ciphertext. More formally, the 16 bytes of plaintext $p_{0,\ldots,}\ p_{15}$ are first split into 4 words $P_{0,\ldots\ldots,}P_3$ of 32 bits each using the little-endian convention and not big-endian convention. In cryptography little endian method is widely used.

$$P_i = \sum_{j=0}^{3} P(4_{i+j}).2^{8j} \qquad i = 0,\ldots,3$$

These words are xored with 4 words of the expanded key, in the input whitening steps.

$$R_{o,i} = P_i \oplus K_i \qquad i = 0,\ldots,3$$

Furthermore, in each of the 16 rounds, the first two words are used as input to the function F, which also takes the round number as input. The third word is xored with the first output of F and then rotated right by one bit. The fourth word is rotated left by one bit and then xored with the second output word of F. Finally, the two halves are exchanged. Thus,

$$\begin{aligned}(F_{r,0}, F_{r,1}) &= F(R_{r,0}, R_{r,1}, r)\\ R_{r+1,0} &= ROR(R_{r,2} \oplus F_{r,0}, 1)\\ R_{r+1,1} &= ROL(R_{r,3}, 1) \oplus F_{r,1}\\ R_{r+1,2} &= R_{r,0}\\ R_{r+1,3} &= R_{r,1}\end{aligned}$$

for r = 0,…., 15 and where ROR and ROL are functions that rotate their first argument (a 32-bit word) left or right by the number of bits indicated by their second argument. The output whitening step undoes the `swap' of the last round, and xors the data words with 4 words of the expanded key.

$$C_i = R_{16,(i+2) \bmod 4} \oplus K_{i+4} \qquad i = 0,\ldots,3$$

In addition to that, the four words of ciphertext are then written as 16 bytes c_0,……, c_{15} using the same little-endian conversion used for the plaintext.

$$c_i = \frac{C[i/4]}{2^{8(i \bmod 4)}} \bmod 2^8 \quad i = 0,\ldots,15$$

2.4.1 The Function F

The function F is the most important part of the encryption algorithm. It does almost 90% of the mixing, transformation and other mathematical operations in the algorithm. It is very important to have a very thorough understanding of the function before proceeding to any implementation. The function has many subparts that needs detail understanding. Basically the function F is a key-dependent permutation on 64-bit values. It takes three arguments, two input words R_0 and R_1, and the round number r used to select the appropriate subkeys. R_0 is passed through the g function, which yields T_0. R_1 is rotated left by 8 bits and then passed through the g function to yield $T1$. The results T_0 and T_1 are then combined in a PHT and two words of the expanded key are added.

$$\begin{aligned}
T_0 &= g(R_0) \\
T_1 &= g(ROL(R_1,8)) \\
F_0 &= (T_0 + T_1 + K_{2r+8}) \bmod 2^{32} \\
F_1 &= (T_0 + 2T_1 + K_{2r+9}) \bmod 2^{32}
\end{aligned}$$

where (F_0, F_1) is the result of F. We can define the function F_0 for use in the analysis. F_0 is identical to the F function, except that it does not add any key blocks to the output. (The Pseudo Hadamard Transform is still performed.).

2.4.2 The Function g

Within the F function lies the g function. In fact the g function is probably the most important element in the F- function. The function g forms the heart of Twofish. The input word X is split into four bytes. Each byte is run through its own key-dependent S-box. Each S-box is bijective, takes 8 bits of input, and produces 8 bits of output. The four results are interpreted as a vector of length 4 over GF (2^8), and multiplied by the 4 x 4 MDS matrix (using the field GF (2^8) for the computations). The resulting vector is interpreted as a 32-bit word which is the result of g.

$$x_i = [X / 2^{8i}] \bmod 2^8 \qquad i = 0,\ldots,3$$

$$y_i = s_i[x_i] \qquad i = 0,\ldots,3$$

$$\begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} & MDS & \\ & & \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

$$Z = \sum_{i=0}^{3} z_i . 2^{8i}$$

where s_i is the key-dependent S-boxes and Z is the result of g. For this to be well-defined, correspondence between byte values and the field elements of GF (2^8) need to be specified. GF (2^8) is represented as GF (2) [x] =$v(x)$ where $v(x) = x^8+x^6+x^5+x^3+1$is a primitive polynomial of degree 8 over GF (2).The field element

$$a = \sum_{i=0}^{7} a_i, x^i \text{ with } a_i \in GF(2)$$

is identified with the byte value

$$a = \sum_{i=0}^{7} a_i, 2^i$$

Besides that, addition in GF (2^8) corresponds to a xor of the bytes. This is some sense of natural mapping.. The MDS matrix is given by:

$$MDS = \begin{pmatrix} 01 & EF & 5B & 5B \\ 5B & EF & EF & 01 \\ EF & 5B & 01 & EF \\ EF & 01 & EF & 5B \end{pmatrix}$$

where the elements have been written as hexadecimal byte values using the above defined correspondence.

2.4.3 The Key Schedule

Another very important building block is the Key Schedule. Complexity often depends on the length of the key. The key schedule has to provide 40 words of expanded key *K*0,……, *K39*, and the 4 key-dependent S-boxes used in the g

function. Twofish is defined for keys of length $N = 128$, $N = 192$, and $N = 256$. Keys of any length shorter than 256 bits can be used by padding them with zeroes until the next larger defined key length. We define $k = 64$. The key M consists of $8k$ bytes $m_0,\ldots,m_{8k-1}$. The bytes are first converted into $2k$ words of 32 bits each

$$M_i = \sum_{j=0}^{3} m_{(4i+j)}.2^{8j} \qquad i = 0,\ldots,2k-1$$

and then into two word vectors of length k.

$$M_e = (M_0, M_2, \ldots, M_{2k-2})$$
$$M_o = (M_1, M_3, \ldots, M_{2k-1})$$

A third word vector of length k is also derived from the key. This is done by taking the key bytes in groups of 8, interpreting them as a vector over GF (2^8), and multiplying them by a 4 x 8 matrix derived from an RS code. Each result of 4 bytes is then interpreted as a 32-bit word. These words make up the third vector.

$$\begin{pmatrix} S_{i,0} \\ S_{i,1} \\ S_{i,2} \\ S_{i,3} \end{pmatrix} = \begin{pmatrix} & \cdots & \cdots & \\ \vdots & & & \vdots \\ \vdots & & RS & \vdots \\ & \cdots & \cdots & \end{pmatrix} \cdot \begin{pmatrix} m_{8i} \\ m_{8i+1} \\ m_{8i+2} \\ m_{8i+3} \\ m_{8i+4} \\ m_{8i+5} \\ m_{8i+6} \\ m_{8i+7} \end{pmatrix}$$

$$S_i = \sum_{j=0}^{3} 8_{ij}.2^{8j}$$

for $i = 0,\ldots,k-1$ and

$$S = (S_{k-1}, S_{k-2}, \ldots, S_0)$$

It is very important to note that S lists the words in "reverse" order. For the RS matrix multiply, GF (2^8) is represented by GF (2) [x] =$w(x)$, where $w(x)$ =

$x^8+x^6+x^3+x^2+1$ is another primitive polynomial of degree 8 over GF (2). The mapping between byte values and elements of GF (2^8) uses the same definition as used for the MDS matrix multiply. Using this mapping, the RS matrix is given by:

$$RS = \begin{pmatrix} 01 & A4 & 55 & 87 & 5A & 58 & DB & 9E \\ A4 & 56 & 82 & F3 & 1E & C6 & 68 & E5 \\ 02 & A1 & FC & C1 & 47 & AE & 3D & 19 \\ A4 & 55 & 87 & 5A & 58 & DB & 9E & 03 \end{pmatrix}$$

Now we have 3 of the most important vectors .The three vectors M_e, M_o, and S form the basis of the key schedule.

2.4.3.1 Additional Key Lengths

Twofish encryption algorithm is a very flexible algorithm. Twofish can accept keys of any byte length up to 256 bits. For key sizes that are not defined above, the key is padded at the end with zero bytes to the next larger length that is defined. For example, an 80-bit key $m_0,\ldots, m_9$ would be extended by setting $m_i = 0$ for $i = 10,\ldots\ldots, 15$ and treating it as a 128-bit key.

2.4.3.2 The Function h

Another very important and subset of the F-function is the h-function. The figure below shows an overview of the function h. This is a function that takes two inputs - a 32-bit word X and a list $L = (L_0,\ldots..,L_{k-1})$ of 32-bit words of length k - and produces one word of output. This function works in k stages. In each stage, the four bytes are each passed through a fixed S-box, and xored with a byte derived from the list. Finally, the bytes are once again passed through a fixed S -box and the four bytes are multiplied by the MDS matrix just as in g. More formally: we split the words into bytes.

$$l_{i,j} = [L_i / 2^{8j}] \bmod 2^8$$
$$x_j = [X / 2^{8j}] \bmod 2^8$$

for $i = 0,\ldots,k-1$ and $j = 0,\ldots,3$. Then the sequence of substitutions and xors is applied.

$$y_{k,j} = x_j \qquad j = 0,\ldots,3$$

$$\begin{aligned}
y_0 &= q_1[q_0[q_0[y_{2,0}] \oplus l_{1,0}] \oplus l_{0,0}] \\
y_1 &= q_0[q_0[q_1[y_{2,1}] \oplus l_{1,1}] \oplus l_{0,1}] \\
y_2 &= q_1[q_1[q_0[y_{2,2}] \oplus l_{1,2}] \oplus l_{0,2}] \\
y_3 &= q_0[q_1[q_1[y_{2,3}] \oplus l_{1,3}] \oplus l_{0,3}]
\end{aligned}$$

Furthermore, q_0 and q_1 are fixed permutations on 8-bit values that will be defined shortly. The resulting vector of y_i's is multiplied by the MDS matrix, just as in the g function.

$$\begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} & \cdots & \cdots & \\ \vdots & & & \vdots \\ \vdots & & MDS & \vdots \\ & \cdots & \cdots & \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

$$Z = \sum_{i=0}^{3} z_i . 2^{8i}$$

where Z is the result of h

If k = 4 we have

$$\begin{aligned}
y_{3,0} &= q_1[y_{4,0}] \oplus l_{3,0} \\
y_{3,1} &= q_0[y_{4,1}] \oplus l_{3,1} \\
y_{3,2} &= q_0[y_{4,2}] \oplus l_{3,2} \\
y_{3,3} &= q_1[y_{4,3}] \oplus l_{3,3}
\end{aligned}$$

If k >= 3 we have

$$\begin{aligned}
y_{2,0} &= q_1[y_{3,0}] \oplus l_{2,0} \\
y_{2,1} &= q_1[y_{3,1}] \oplus l_{2,1} \\
y_{2,2} &= q_0[y_{3,2}] \oplus l_{2,2} \\
y_{2,3} &= q_0[y_{3,3}] \oplus l_{2,3}
\end{aligned}$$

2.4.3.3 Key-dependent S-boxes

Key dependent S-boxes are very important in cryptography. They prevent the attackers from knowing what is in the S-boxes. It is well understood that the

complexity of the keyed S-boxes depends on the length of the key. S-boxes in the function *g can be defined by*

$$g(X) = h(X, S)$$

That is, for $i = 0,\ldots..,\ 3$, the key-dependent S-box s_i is formed by the mapping from *xi* to y_i in the *h* function, where the list *L* is equal to the vector *S* derived from the key. The downside of this operation is that it takes longer to set up for a key since S-boxes have to be built for each key.

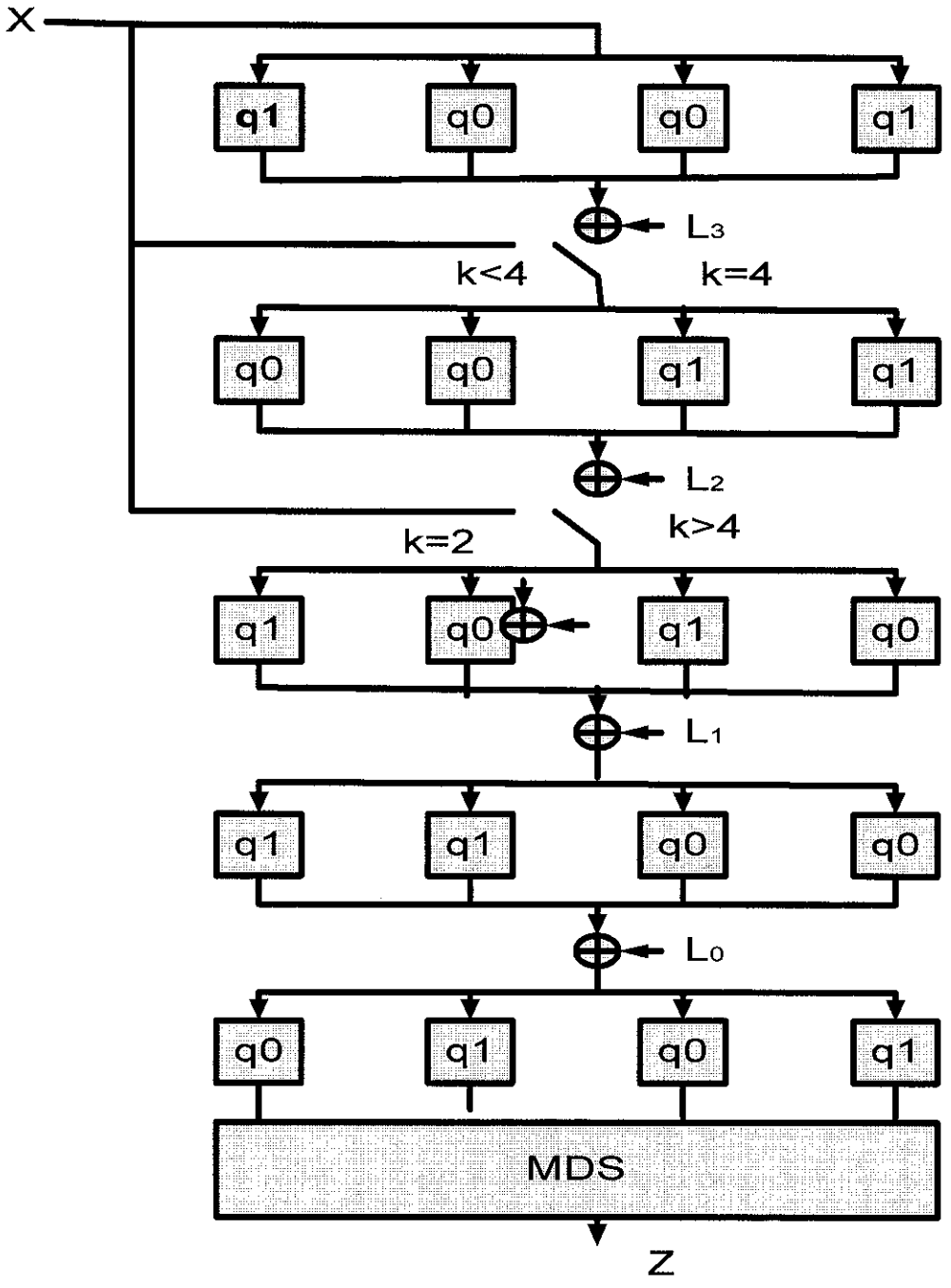


Figure 2: Function h

2.4.3.4 The Expanded Key Words K_j

The words of the expanded key are defined using the *h* function.

$$\rho = 2^{24} + 2^{16} + 2^8 + 2^0$$
$$A_i = h(2i\rho, M_e)$$
$$B_i = ROL(h((2i+1)\rho, M_o), 8)$$
$$K_{2i} = (A_i + B_i) \bmod 2^{32}$$
$$K_{2i+1} = ROL((A_i + 2B_i) \bmod 2^{32}, 9)$$

The constant ρ is used here to duplicate bytes; it has the property that for i = 0,..., 255, the word i ρ consists of four equal bytes, each with the value i. The function h is applied to words of this type. For Ai the byte values are $2i$, and the second argument of h is Me. Bi is computed similarly using $2i + 1$ as the byte value and Mo as the second argument, with an extra rotate over 8 bits. The values Ai and Bi are combined in a PHT. One of the results is further rotated by 9 bits. The two results form two words of the expanded key.

2.4.3.5 The Permutations q_0 and q_1

Permutations are very important in cryptography. The permutations q_0 and q_1 are fixed permutations on 8-bit values. They are constructed from four different 4-bit permutations each. For the input value x, the corresponding output value y *is defined* as follows:

$$a_0, b_0 = [x/16], x \bmod 16$$
$$a_1 = a_0 \oplus b_0$$
$$b_1 = a_0 \oplus ROR_4(b_{0,1}) \oplus 8a_0 \bmod 16$$
$$a_2, b_2 = t_0[a_1], t_1[b_1]$$
$$a_3 = a_2 \oplus b_2$$
$$b_3 = a_2 \oplus ROR_4(b_{2,1}) \oplus 8a_2 \bmod 16$$
$$a_4, b_4 = t_2[a_3], t_3[b_3]$$
$$y = 16b_4 + a_4$$

where ROR4 is a function similar to ROR that rotates 4-bit values. First, the byte is split into two nibbles. These are combined in a bijective mixing step. Each nibble is then passed through its own 4-bit fixed S-box. This is followed

by another mixing step and S-box lookup. Finally, the two nibbles are recombined into a byte. For the permutation q_0 the 4-bit S-boxes are given by

t0 = [8 1 7 D 6 F 3 2 0 B 5 9 E C A 4]
t1 = [E C B 8 1 2 3 5 F 4 A 6 7 0 9 D]
t2 = [B A 5 E 6 D 9 0 C 8 F 3 2 4 7 1]
t3 = [D 7 F 4 1 2 6 E 9 B 3 0 8 5 C A]

where each 4-bit S-box is represented by a list of the entries using hexadecimal notation. (The entries for the inputs 0, 1,... ,15 are listed in order.) Similarly, for q_1 the 4-bit S-boxes are given by

t0 = [2 8 B D F 7 6 E 3 1 9 4 0 A C 5]
t1 = [1 E 2 B 4 C 3 7 6 D A 5 F 9 0 8]
t2 = [4 C 7 5 1 6 9 A 0 E D 8 2 B 3 F]
t3 = [B 9 5 1 C 3 D E 6 4 7 F 2 0 8 A]

One interesting point to note here is the usage of the 1 bit rotation. The technique is widely employed in the field of cryptography and Twofish round to break up the byte aligned nature of the operations. Each of the 4 32-bit quantities in the block is used once in each of the 8 possible bit positions (mod 8).

2.4.4 Round Function Overview

The figure below shows a more detailed view of how the function F is computed each round when the key length is 128 bits. Incorporating the S-box and round subkeygeneration makes the Twofish round function look more complicated, but is useful for visualizing exactly how the algorithm works.

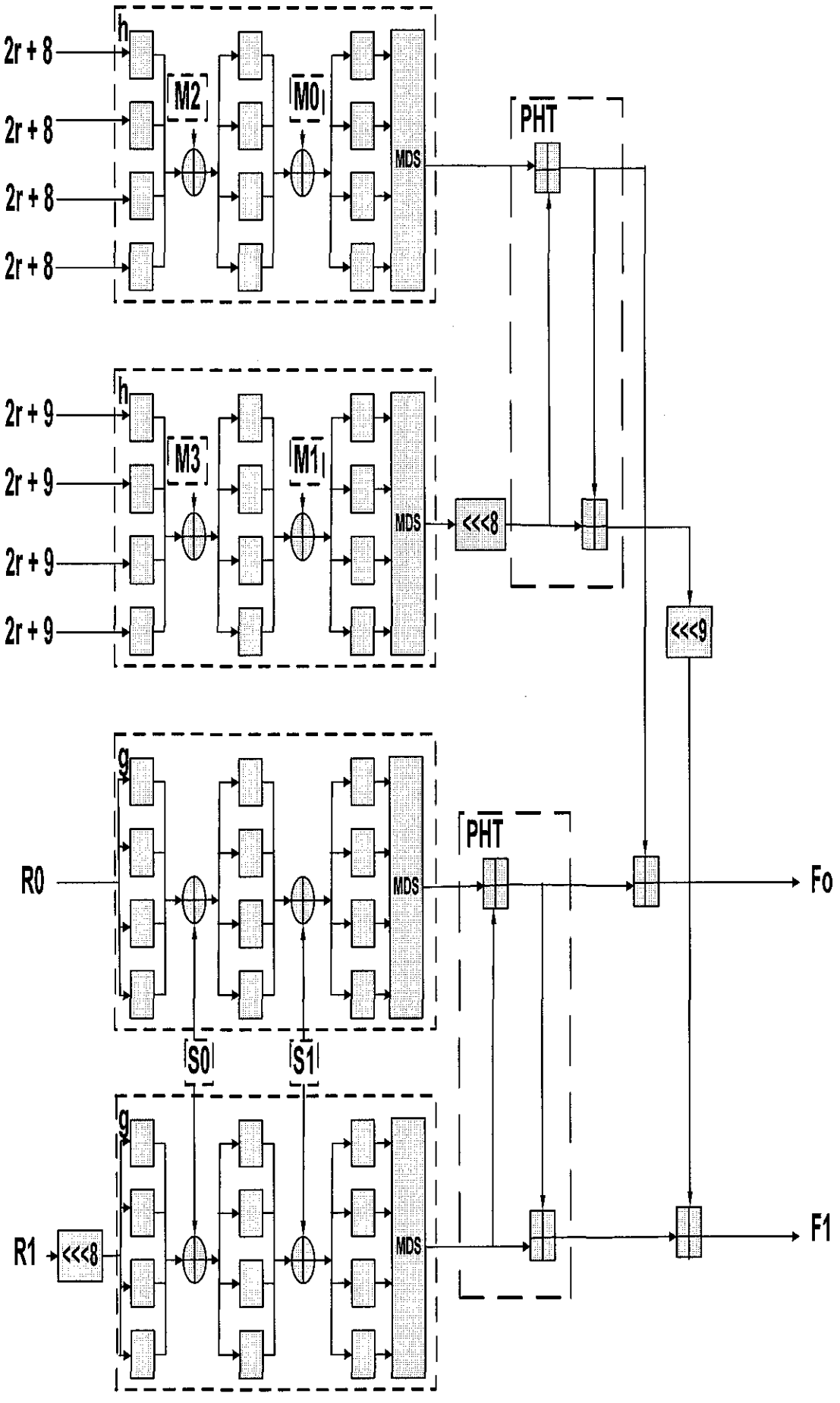


Figure 3: A view of a Single Round F Function (128-bit Key)

CHAPTER 3: METHODOLOGY

3.1 PROCEDURE IDENTIFICATION

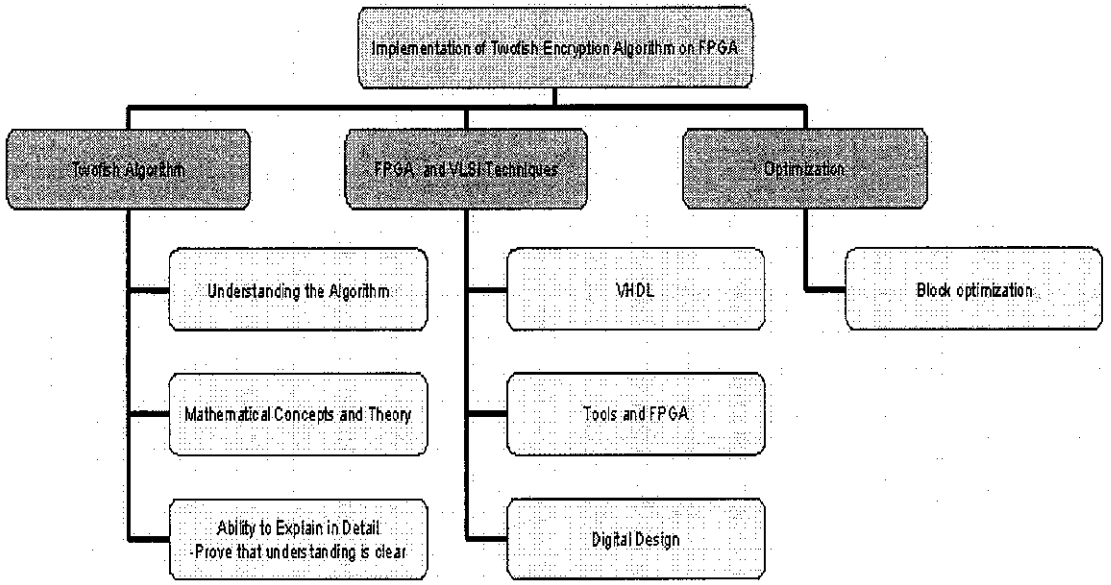


Figure 4: Project Development Block

.

From the chart above, the project is broken into 3 main blocks namely Twofish Algorithm, FPGA and VLSI Techniques, and Optimization. For the project, the short term priority is to understand the algorithm including the mathematical concepts and theory. This is a VERY IMPORTANT step, before the next step is considered. Once the nature of the algorithm is fully understood, implementation of the other steps would be more straightforward. The medium term and long term objective is to incorporate both the learning of VHDL, understanding the tools and FPGA as well combining the digital design with the block optimization.

3.2 TOOLS

The tools required are as follows:

- VHDL Simulator – ALDEC
- FPGA Synthesizer - XILINX Synthesizing Tools
- FPGA – **Spartan2 - XC2S200-5PQ208**

CHAPTER 4: PROJECT WORK

This is the heart of the report. Basically the project work is divided into 2 main categories:-

- **Design Implementation** – This section all the design decisions that have been made to implement the project. It covers the major building blocks of the design and how the various blocks have been integrated to achieve an overall complete design.
- **Programming Strategy** – This section covers how the whole programming work started with a top to down approach. It only explains the top level blocks without going deep into each block. The justification is, each block consists of multi level blocks and it is too long to explain in this report.

4.1 DESIGN IMPLEMENTATION

Over the period of implementation of this project, 2 different designs with different initial objectives were implemented. The first design was implemented in the first semester and the second design was implemented in the second semester. The general objectives of both designs are described below:-

- **Design 1**
  - Minimum hardware resources usage
  - **Using a single F-Function (modified version)**
  - Optimized design with reasonable latency, throughput and throughput per gate.
- **Design 2**
  - Reasonably minimum hardware resources usage
  - **Using 4 units of modified F-Function of Design 1**
  - Very small latency
  - Very high throughput
  - Very high throughput per gate

Besides that, decision has also been made to stick to a general design architecture (Figure 1) but modifying the F-Function accordingly to the objectives of **Design 1** and **Design 2.**The design implantation covers 2 main parts namely design decisions and also integration and overall structure.

Note: The explanation begins with the common design modules and where necessary the differences between Design 1 and Design 2 would be highlighted.

4.1.1 Design Decisions

The Twofish structure offers a great deal of flexibility in terms of space versus speed tradeoffs. I have decided to go for a minimum hardware implementation of the algorithm with hopes of fitting the circuit on Xilinx Spartan 200 FPGA. Among the design decisions are as follows: -

- One h-function, instead of 2 would be used in computing the K-subkeys and the encrypted data (Refer Figure 2). This means the modified F- Function would only consist of 1 h-function and not 2 h-functions as proposed by the author. **(Applicable to Design 1 only)**
- Uses 4 individual units of h-functions whereby in 1 F-function there is only 1 h-function. In other words, 4 units of modified F-function of Design 1 would be used to construct 1 major unit of F-Function of Design 2. **(Applicable to Design 2 only)**
- Zero keying would be implemented as it would be a better choice as the RAM needed would consume too much space on the FPGA.
- An ecryptor/decryptor would be implemented.

4.1.1.1 Building Blocks

The common major building blocks for both designs explained here are as follows:-

- Q-Permutations
- S-boxes
- Maximum Distance Separable (MDS)
- Reed-Solomon Matrix
- Operation Selector

4.1.1.1.1 Q-Permutations

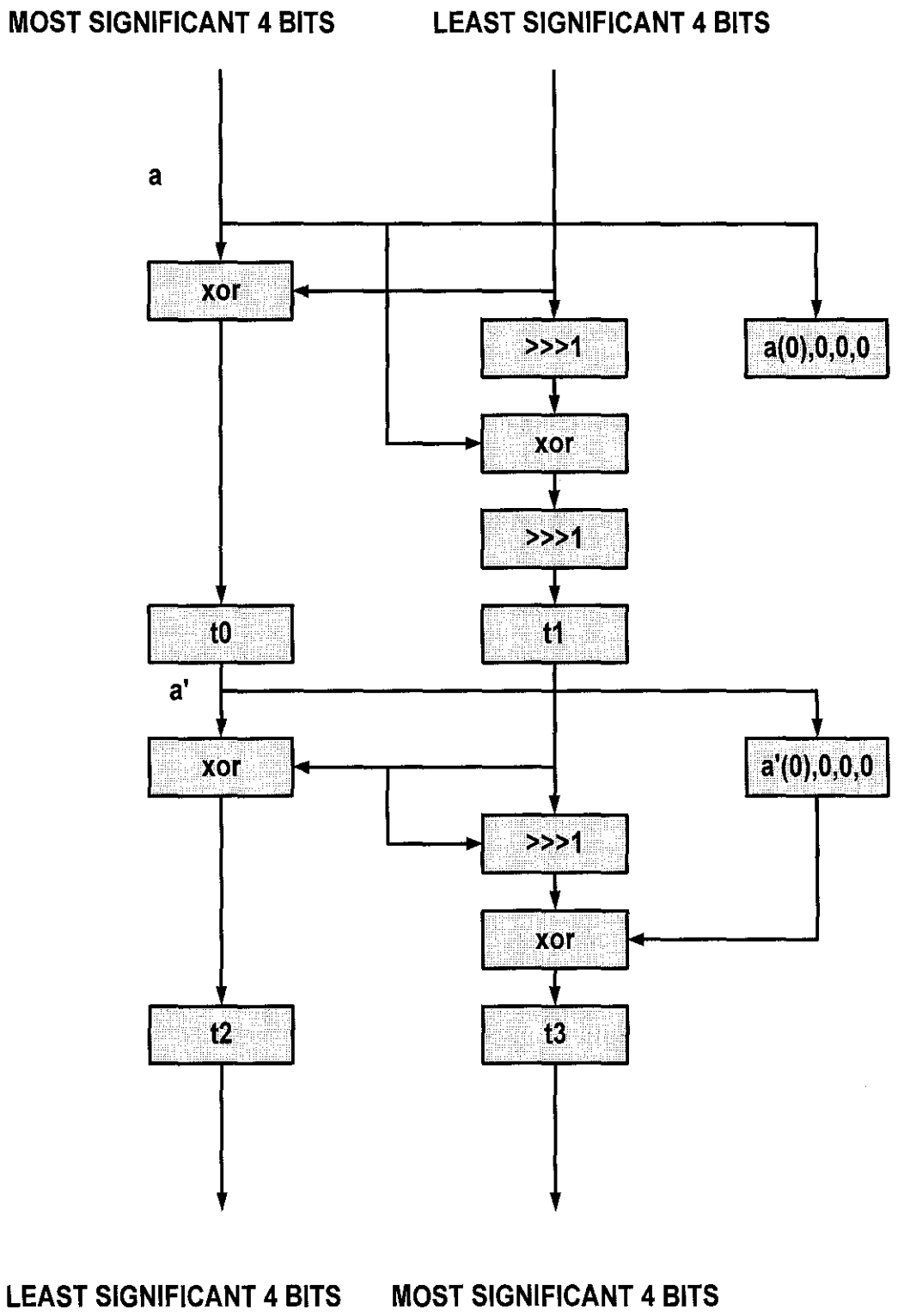


Figure 5: Q-Permutation

The Q-Permutation is one of the most important elements at the core of the design of Twofish. Lookup tables are involved in this part. The permutation is executed on a byte of input, which is split in two before being rotated and modified along different paths. The most important operations of the permutation are executed with four lookup tables labeled t0, t1, t2 and t3. Each lookup table takes 4 bits of input and produces a 4-bit value. Each lookup table thus has 16 entries of 4 bits. There are four lookup tables per q permutation and two different q-permutations, q0 and q1, each with its set of lookup table. The implementation of the q permutations especially the lookup tables could have been implemented using the ROM but it would have taken up a lot of space. This is because a large number of q permutations have to operate in parallel. Thus a separate ROM would have needed for each instance of the q permutation. To save space and make it more efficient, the lookup tables were implemented using logic that is using behavioral VHDL.

4.1.1.1.2 S-Boxes

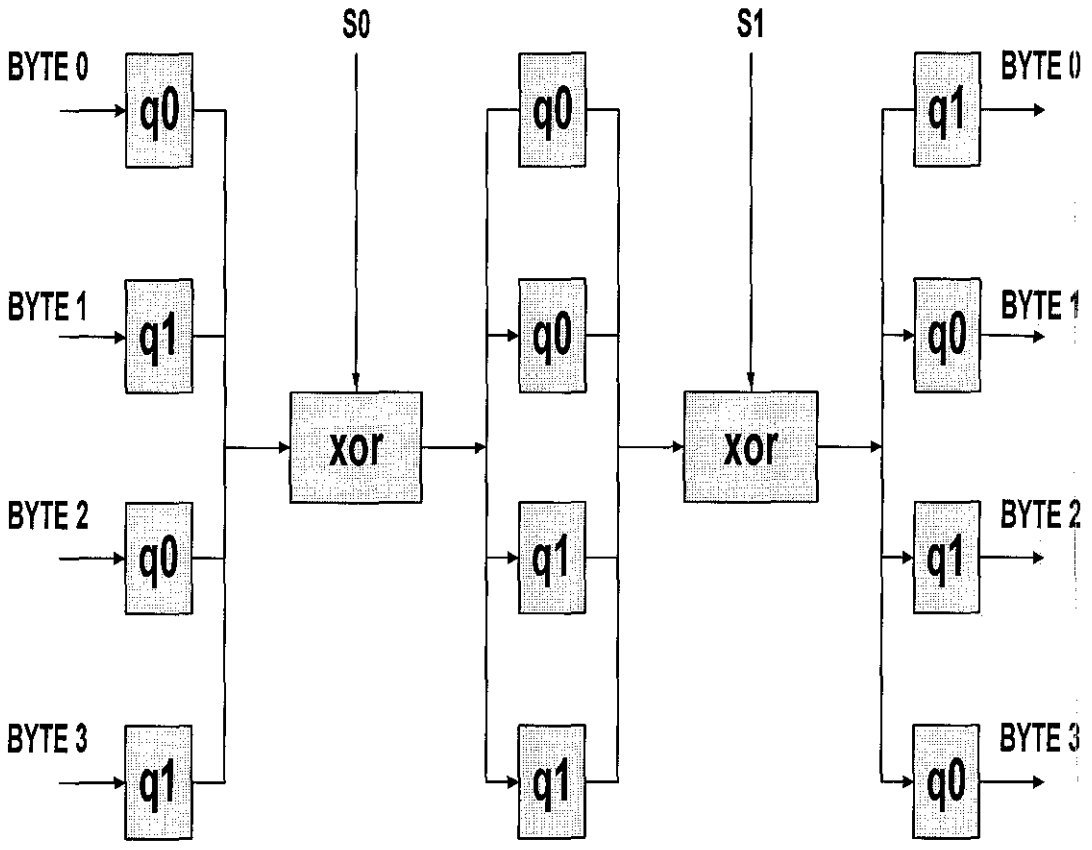


Figure 6: S-boxes

The S-Box operates on a 32-bit word. Each byte of the word passes through three Q-Permutations. This is evident by the figure above. The output of each bank of q-permutations is then recombined into a word and XOR-ed with a 32-bit value [7]. These two values are derived from the key material and Twofish's s-boxes are thus referred to as "key-dependent" s-boxes.

4.1.1.1.3 Maximum Distance Separable Matrix

$$MDS = \begin{pmatrix} 01 & EF & 5B & 5B \\ 5B & EF & EF & 01 \\ EF & 5B & 01 & EF \\ EF & 01 & EF & 5B \end{pmatrix}$$

Figure 7: MDS

One of the most important diffusion elements is the Maximum Distance Separable. The Maximum Distance Separable (MDS) Matrix is a 4x4 matrix of bytes that multiplies a vector of four bytes. Multiplications are carried out in the Galois Field GF(2^8) with the primitive polynomial $x^8 + x^6 + x^5 + x^3 + 1$. Each byte is converted into a polynomial in which each power *p* of x is present only if the *p*-th bit is 1. A multiplication in GF amounts to a multiplication of polynomials followed by a division by the primitive polynomial. The result is converted back to a bit vector by setting a bit to 0 if the corresponding power of x has an odd coefficient and 1 otherwise (modulo 2 divisions). In this case the computations are fairly straightforward since there are only three coefficients: 0x01, 0xEF and 0x5B. **The result of a multiplication can be reduced to a series of XOR's for each bit of the output.** For example, multiplying *a* by 5B results in byte *b* [4-5]:

b0 = a2 xor a0

b1 = a3 xor a1 xor a0

b2 = a4 xor a2 xor a1

b3 = a5 xor a3 xor a0

b4 = a6 xor a4 xor a1 xor a0

b5 = a7 xor a5 xor a1

b6 = a6 xor a0

b7 = a7 xor a1

NOTE: We can't continue the coding, without generating these tables because without reducing the expression, the expression can't be implemented on hardware. So it is very important, to generated the tables in a proper sequence and to make sure they are free from errors. If any of the tables contain errors, the encryption/decryption process would not produce correct results.

Refer APPENDIX A, to view the remaining minimization of multiplicand and how the minimization was implemented through a structured table.

4.1.1.1.4 Reed-Solomon Matrix

$$\begin{pmatrix} s_{i,0} \\ s_{i,1} \\ s_{i,2} \\ s_{i,3} \end{pmatrix} = \begin{pmatrix} 01 & A4 & 55 & 87 & 5A & 58 & DB & 9E \\ A4 & 56 & 82 & F3 & 1E & C6 & 68 & E5 \\ 02 & A1 & FC & C1 & 47 & AE & 3D & 19 \\ A4 & 55 & 87 & 5A & 58 & DB & 9E & 03 \end{pmatrix} * \begin{pmatrix} m_{8i} \\ m_{8i+1} \\ m_{8i+2} \\ m_{8i+3} \\ m_{8i+4} \\ m_{8i+5} \\ m_{8i+6} \\ m_{8i+7} \end{pmatrix}$$

Figure 8: Reed-Solomon Matrix

The Reed-Solomon (RS) matrix is also another diffusion element. But the diffusion element takes effect on the key materials and not on the ciphertext (data). The Reed-Solomon (RS) matrix is similar to the MDS matrix. In this case the multiplication is executed between an 8x4 matrix and a vector of 8 bytes. The computations are done in GF (2^8) with a different prime polynomial: $x^8 + x^6 + x^3 + x^2 + 1$. Unlike the MDS matrix, the RS matrix has a large number of different multiplicands. In order to minimize the resources used, a series of XOR's was derived for each of the 23 multiplicands to give equations similar to those seen in the MDS matrix. These equations were not given and had to be derived [6]. **(Refer APPENDIX B)**

NOTE: We can't continue the coding, without generating these tables because without reducing the expression, the expression can't be implemented on hardware. So it is very important, to generated the tables in a proper sequence and to make sure they are free from errors. If any of the tables contain errors, the encryption/decryption process would not produce correct results.

Refer APPENDIX B, to view the remaining minimization of multiplicand and how the minimization was implemented through a structured table.

4.1.1.1.5 Operation Selector

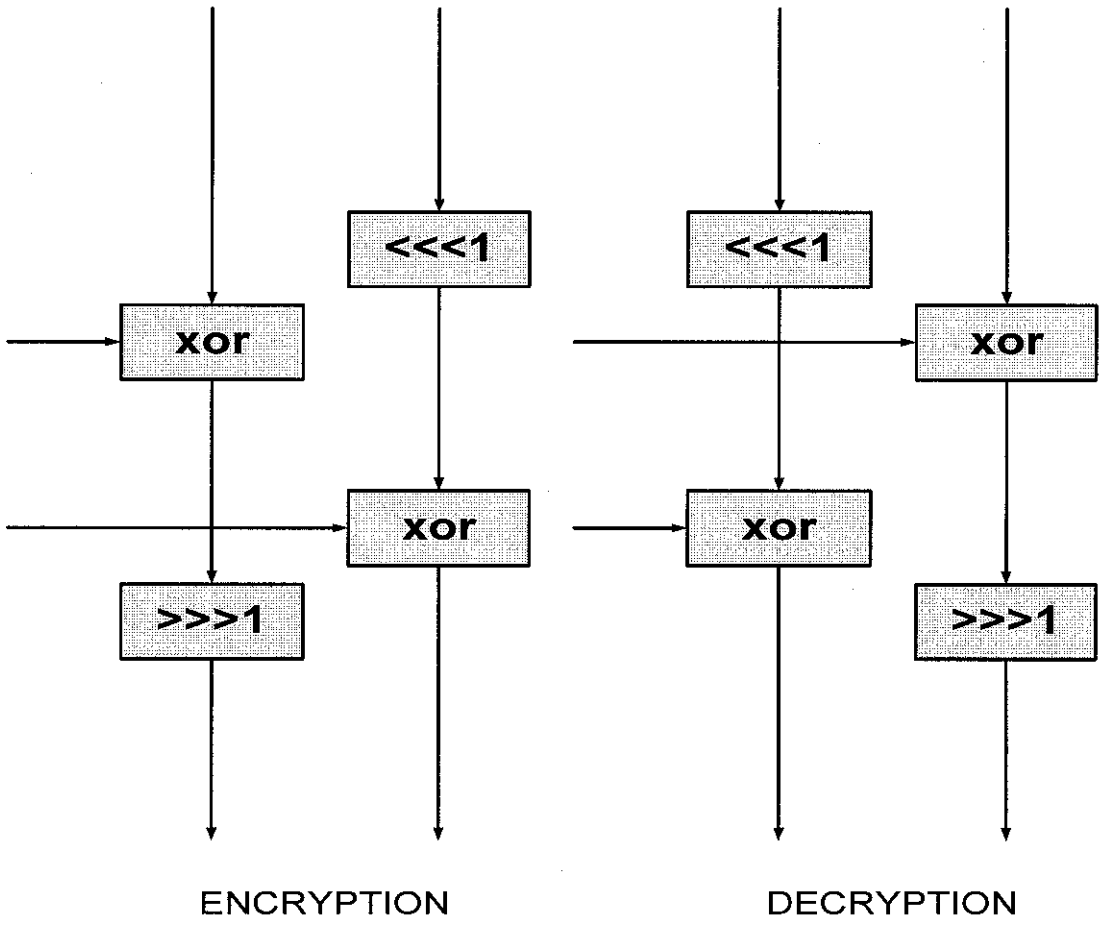


Figure 9: Operation Selector

The authors developed Twofish encryption algorithm to be a very symmetric algorithm. Encryption and decryption can be executed with almost all the

same pieces of hardware. This is evident from the figure above. The first difference is that the sub-keys must be used in reverse order. The second difference is at the output stage of the round. As shown above, the output stage consists of a rotation and an XOR. Basically the block diagram of the encryption is exactly the opposite of the decryption block diagram.

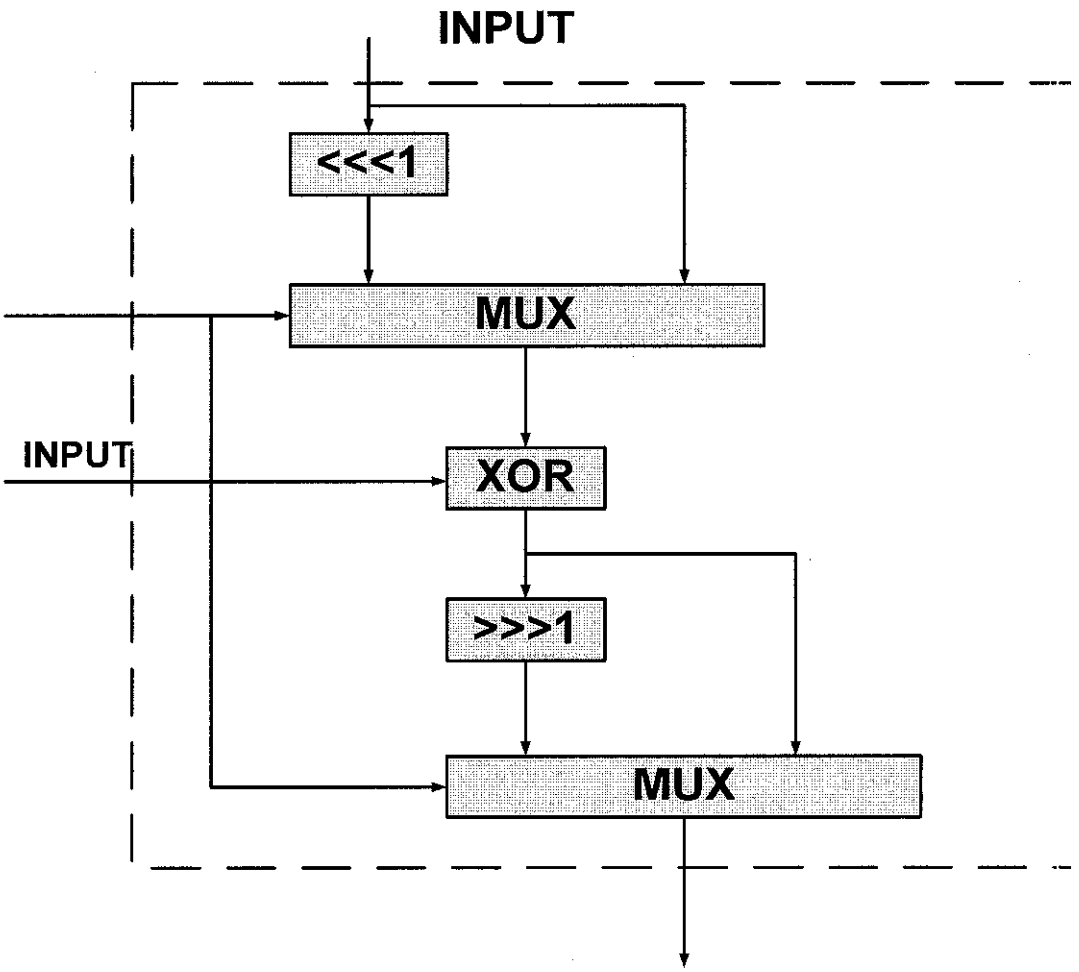


Figure 10: Building Block for Operation Selection

4.1.2 Integration and Overall Structure

The integration and overall structure basically covers the following sections: -

- Input Register Module
- Outer Register Module
- Key Register Module
- Modified F-Function
- Design Overall Structure

- Controller

4.1.2.1 Input Register Module

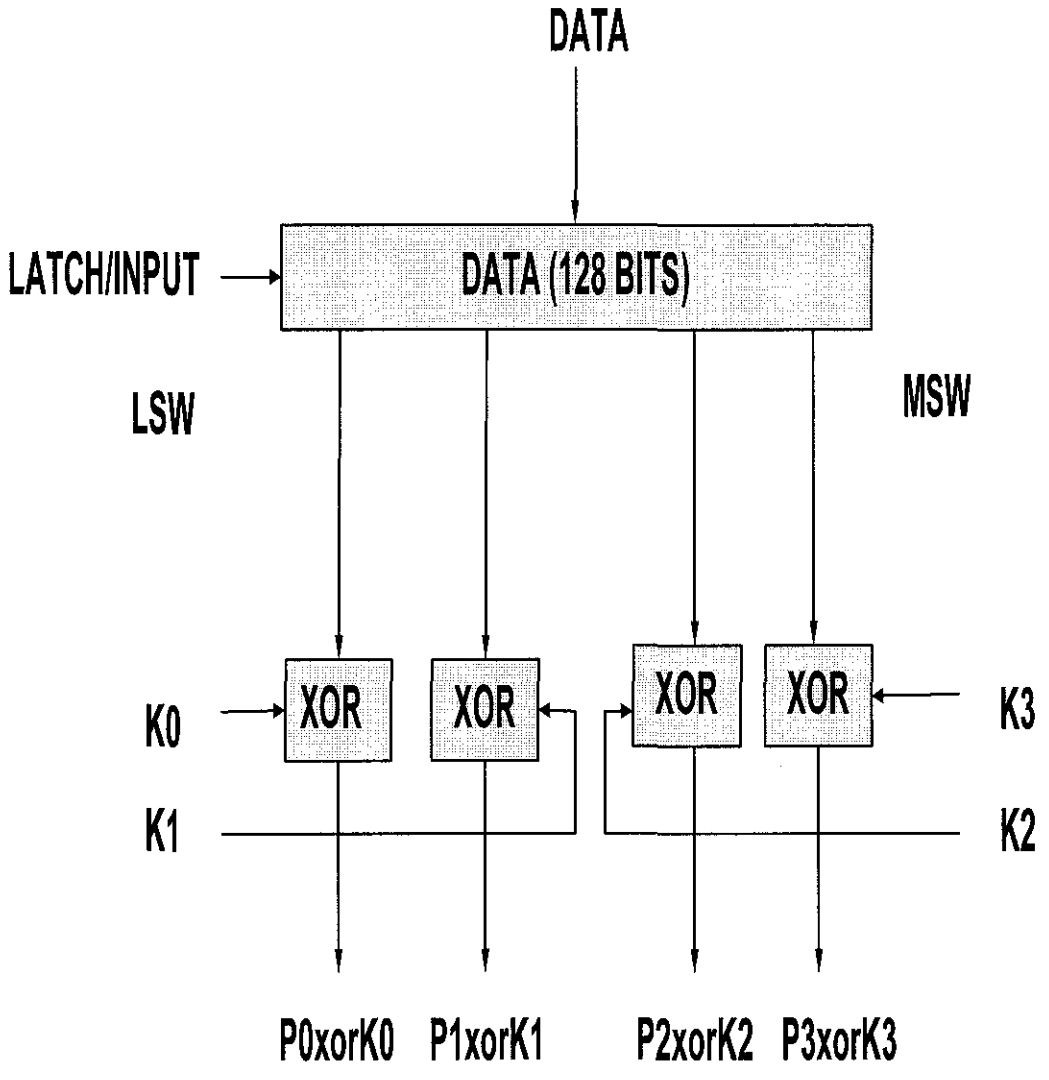


Figure 11: Input Register Module for Design 1

The above diagram shows the input register module. The input register module combines the data register with the input whitening stage. For this project, it was decided to use the data block that is 128 bits wide. A signal is needed to force data to be latched into the register. The input whitening is done asynchronously. Here subkeys are xored with 32 bits data blocks. Basically the output of this module are four 32 bits words. These words corresponds to the input whitening of the 4 words on the input data. This output is valid when the 4 sub-keys are set correctly on the input. Sub-keys K0, K1, K2 and K3 should be used for encryption and K4, K5, K6 and K7 for decryption.

4.1.2.2 Outer Register Module

Design 1

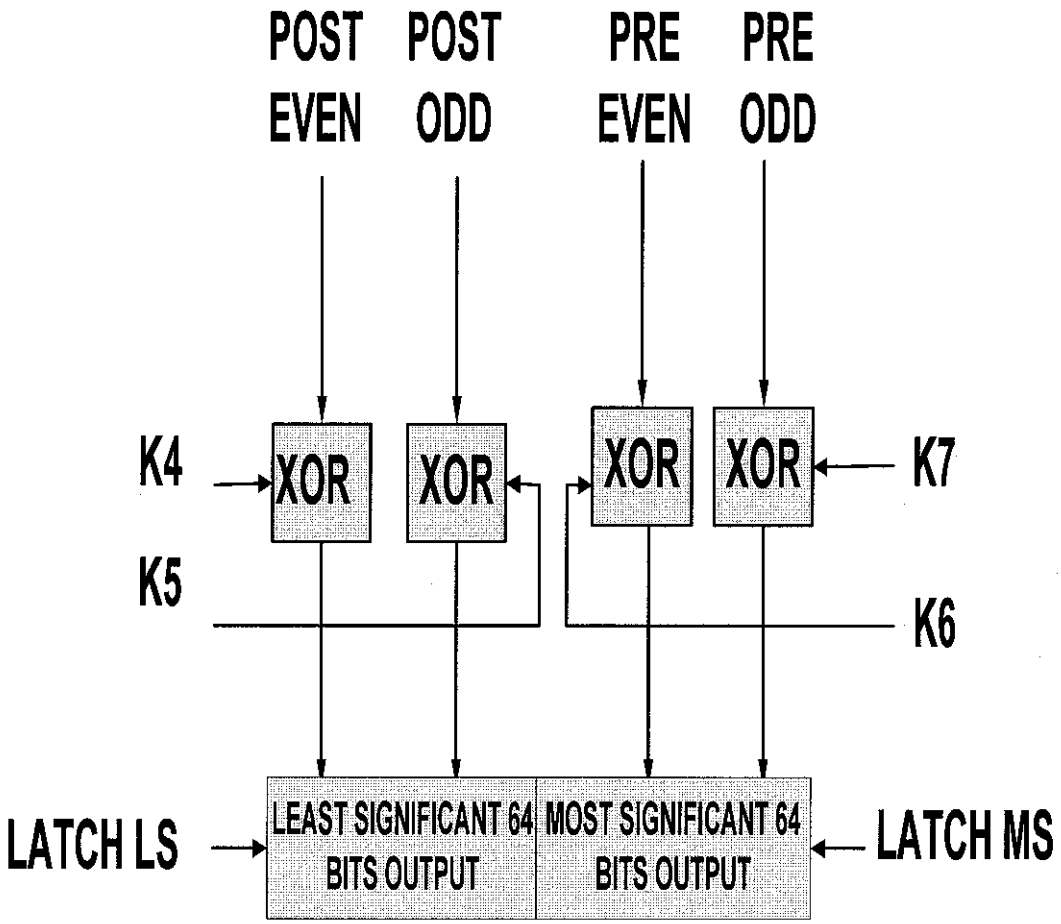


Figure 12: Output Register Module for Design 1

The diagram above shows the output register module. The output register module incorporates the output whitening stage and the output register. Four words of input are XOR-ed with 4 sub-keys to produce the output. The sub-keys must be set to K4, K5, K6 and K7 for encryption and K0, K1, K2 and K3 for decryption. Due to the structure of the cipher only two keys will be available at any time. The output is thus latched in two steps with signals latchLS and latchMS. In short, at any time, two 64 bits of output values are available at the output registers of which only 1 value corresponds to the instantaneous value. This depends on which of the pair of subkeys are fed at that time. (either K4 and K5 or K6 and K7).

Design 2

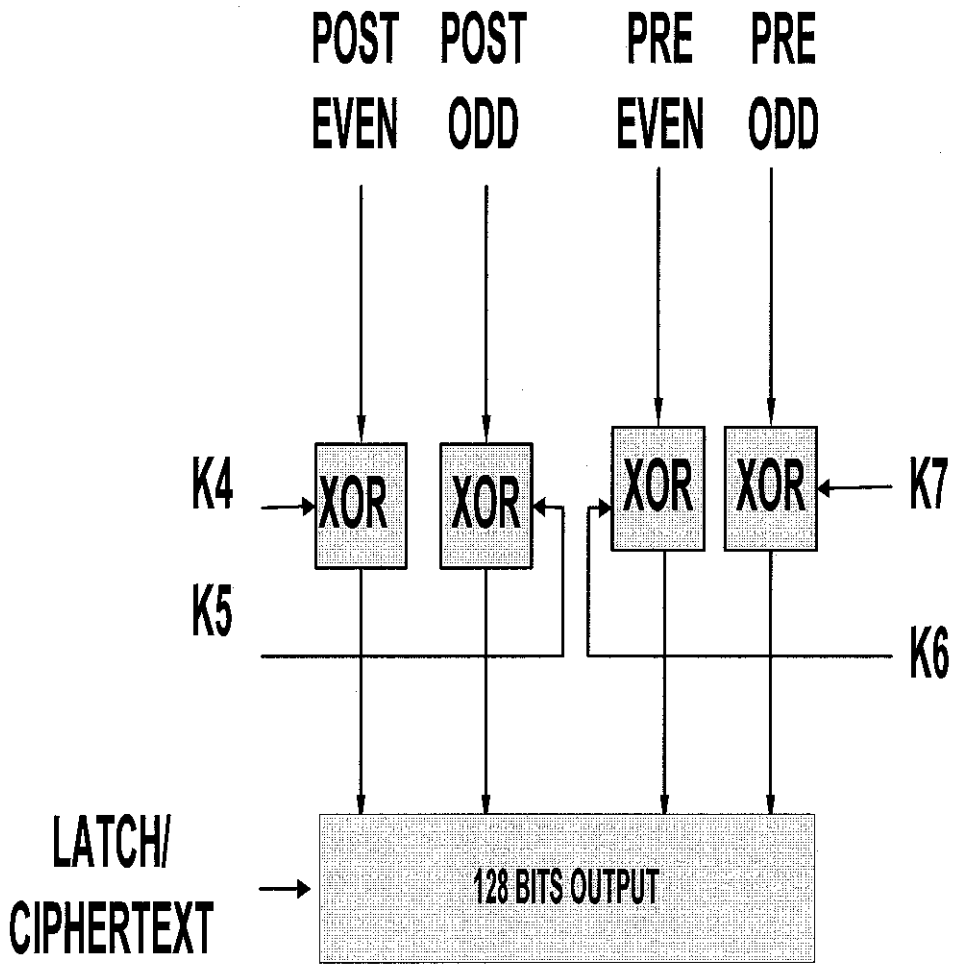


Figure 13: Output Register Module for Design 2

Both Design 1 and Design 2 are quite similar, but the only difference is for Design 2, by sending a LATCH_CIPHERTEXT signal with the presence of the output as mentioned above, the whole complete 128 bit ciphertext will be available at the output register. The benefit is only 1 clock cycle is needed to achieve this result unlike 2 clock cycles for Design 1.

4.1.2.3 Key Register Module

Another very important part in the design consideration is the key register module. The key register module is used to store the 128- bit key material and to produce two derived values (S0 and S1) using the RS matrix. The key is simply latched into the register when the 'Latchkey' signal is raised. The two S-values are computed in two steps by reusing a single RS-matrix. During the

clock cycle in which the key is latched a multiplexer sets the input of the RS matrix to the least-significant double word of the key and the result is stored in the S0 register. At all other times the input is set to the most-significant double word and the output S1 gives the value of S1.

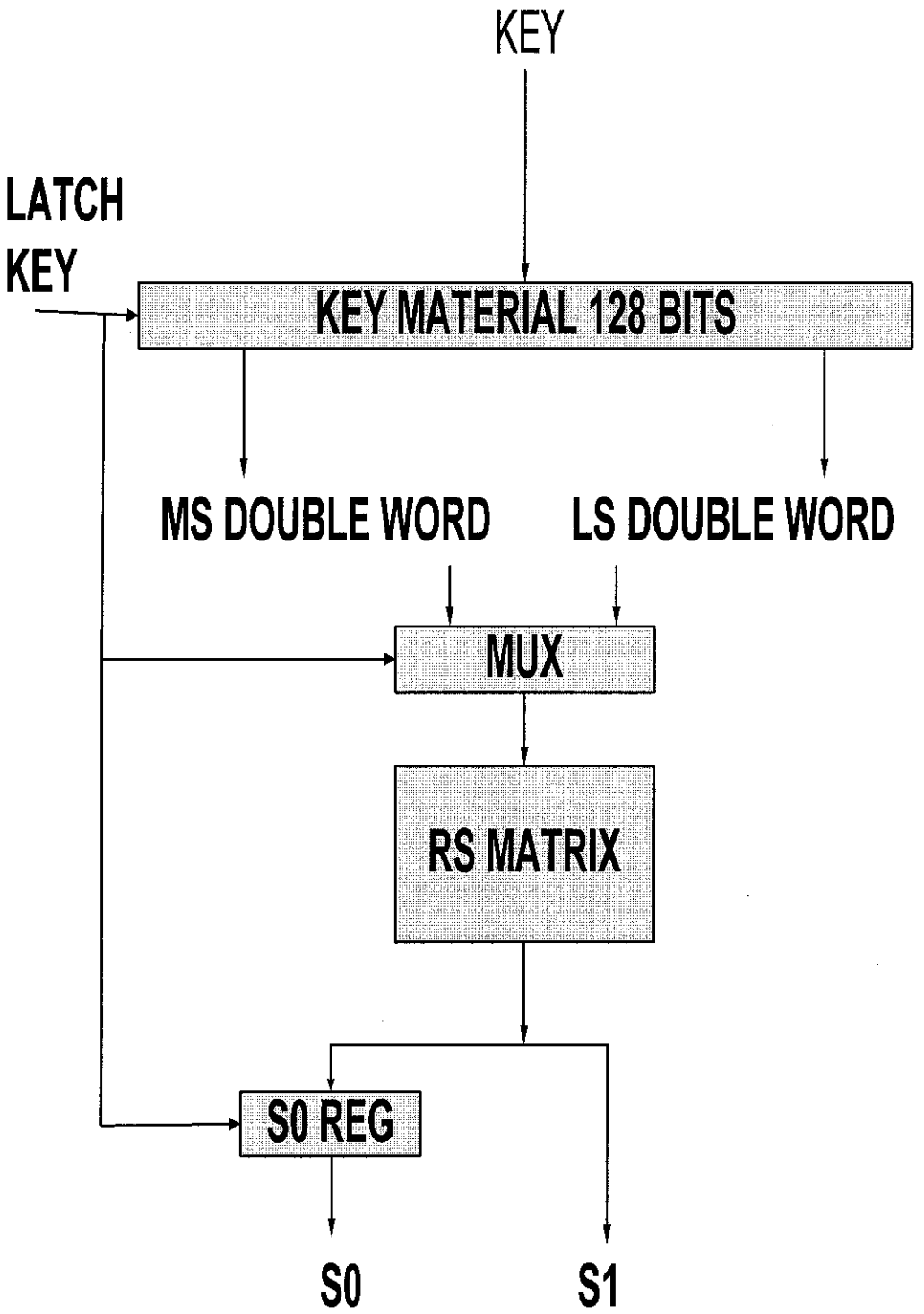


Figure 14: Key Register Module

4.1.2.4 Modified F-Function

Design 1

One of the fundamental design decisions is in the minimizing the hardware resources. As a result the structure of the cipher had to be modified in order to be able to use the same h-function for all computation purposes. The diagram below shows the modified F-function that was designed in order to meet the design requirements. As evident in the diagram above, the input and a rotated version of the input are multiplexed to reproduce both inputs that can be presented at the input of an h-function. Moreover, in order to compute the result of the PHT, two registers had to be added to store the value of the first input or the former while the second was being computed. This registers are needed to temporary hold the values. **Thus, these two registers act as pipeline registers.** The other two multiplexers added are used to choose between the path used for computing a key or the path used to encrypt or decrypt data. The signal INPUT_F could carry either key data or plaintext data. The key data could be even key data or odd key data. The same thing goes for the plaintext data. It could be even plaintext data or odd plaintext data. A complete key cycle takes place when we obtained the signal OUT_EVEN and OUT_ODD. This can only takes place after 2 clock cycles. The reason is, at any clock cycle only one data appears at INPUT_F signal. At this time, the OUT_EVEN is carrying output even key signal and the OUT_ODD refers to output odd key signal. The same thing goes for the plaintext. For a complete plaintext cycle to take place, two clock cycles are needed. Once again, after 2 clock cycles, the OUT_EVEN is carrying output even plaintext signal and the OUT_ODD refers to output odd plaintext signal. As for the overall design, a complete cycle for a round only takes place after 4 clock cycles that is after obtaining the signal values of output even key signal, output odd key signal, output even plaintext signal and output odd plaintext signal.

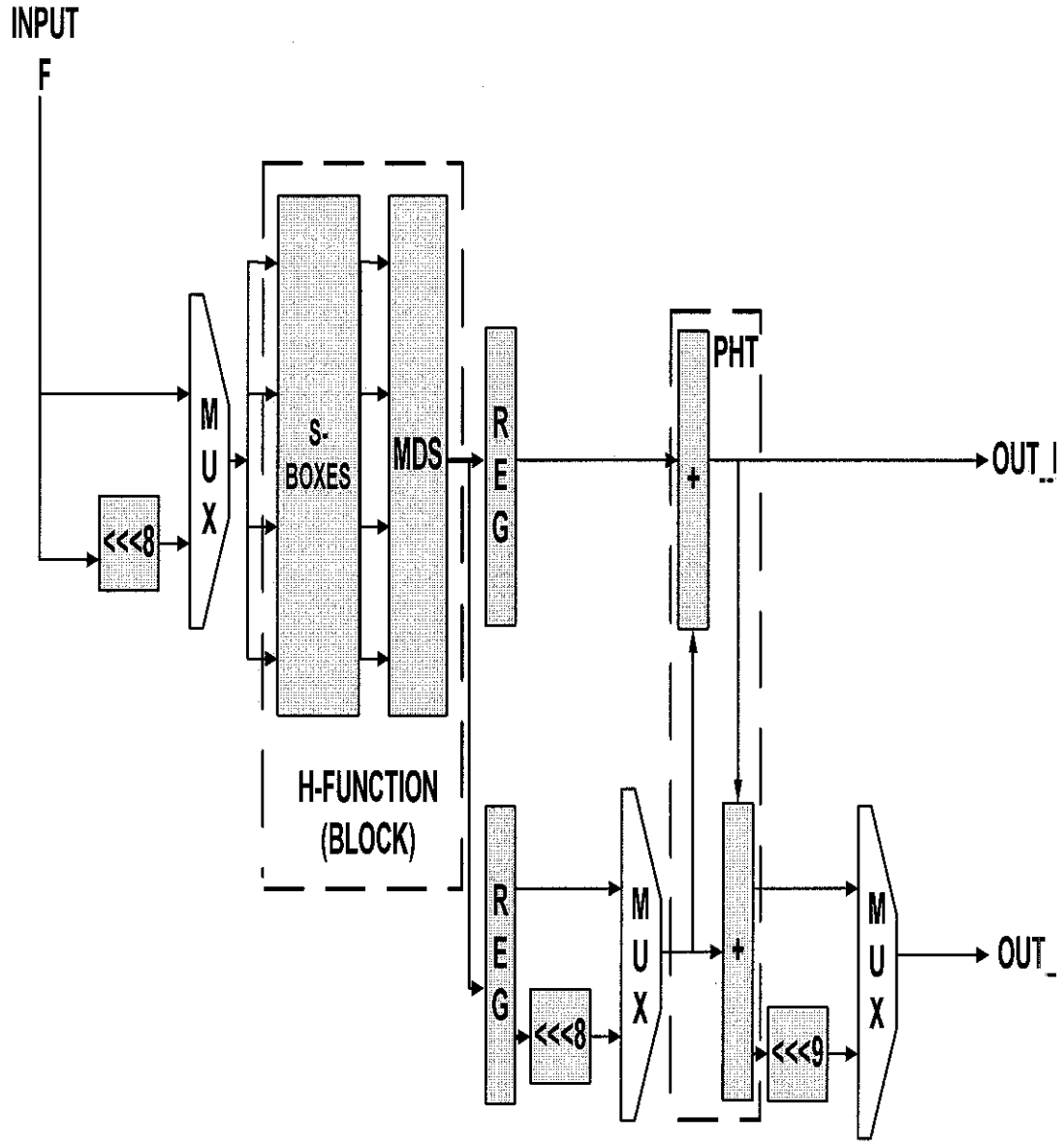


Figure 15: Generalized F-Function of Design 1

Design 2

The modified F-Function of Design 2 has utilized the flexibility of the modified F-Function of Design 1. As can be seen in the diagram, there are 4 input signals namely evenkeygenerator, oddkeygenerator, evenplaintextgenerator and oddplaintextgenerator. This 4 input signals would generate 4 corresponding output signals namely output even key signal, output odd key signal, output even plaintext signal and output odd plaintext signal. Furthermore, the complete cycle for a round process that takes 4 clock cycles to be performed in Design 1 takes only 1 clock cycle in Design 2. Another

interesting point to note is that the lower 2 blocks namely Block 3 and Block 4 could be used to calculate (generate) signals evenkeygenerator and oddkeygenerator. These 2 blocks are special modified blocks that could perform its functions besides performing functions of Block 1 and 2. This could be done by sending the input signals of evenkeygenerator to evenplaintextgenerator and oddkeygenerator to oddplaintextgenerator. These special capabilities make Design 2 a very impressive design because this feature is needed during **input and output whitening processes**. During these processes, only keys are calculated. If we don't have these features, then Block 3 and Block 4 would be idle because they would not be able to calculate the keys. Benefiting from these features, no blocks would be idle anymore thus reducing the latency and increasing the throughput.

NOTE: The term evenplaintextgenerator and oddplaintextgenerator would be used extensively through out the report. The plaintext term that is mentioned here actually refers to intermediate value and not the original value. This is because there are 16 rounds, and the intermediate plaintext value changes.

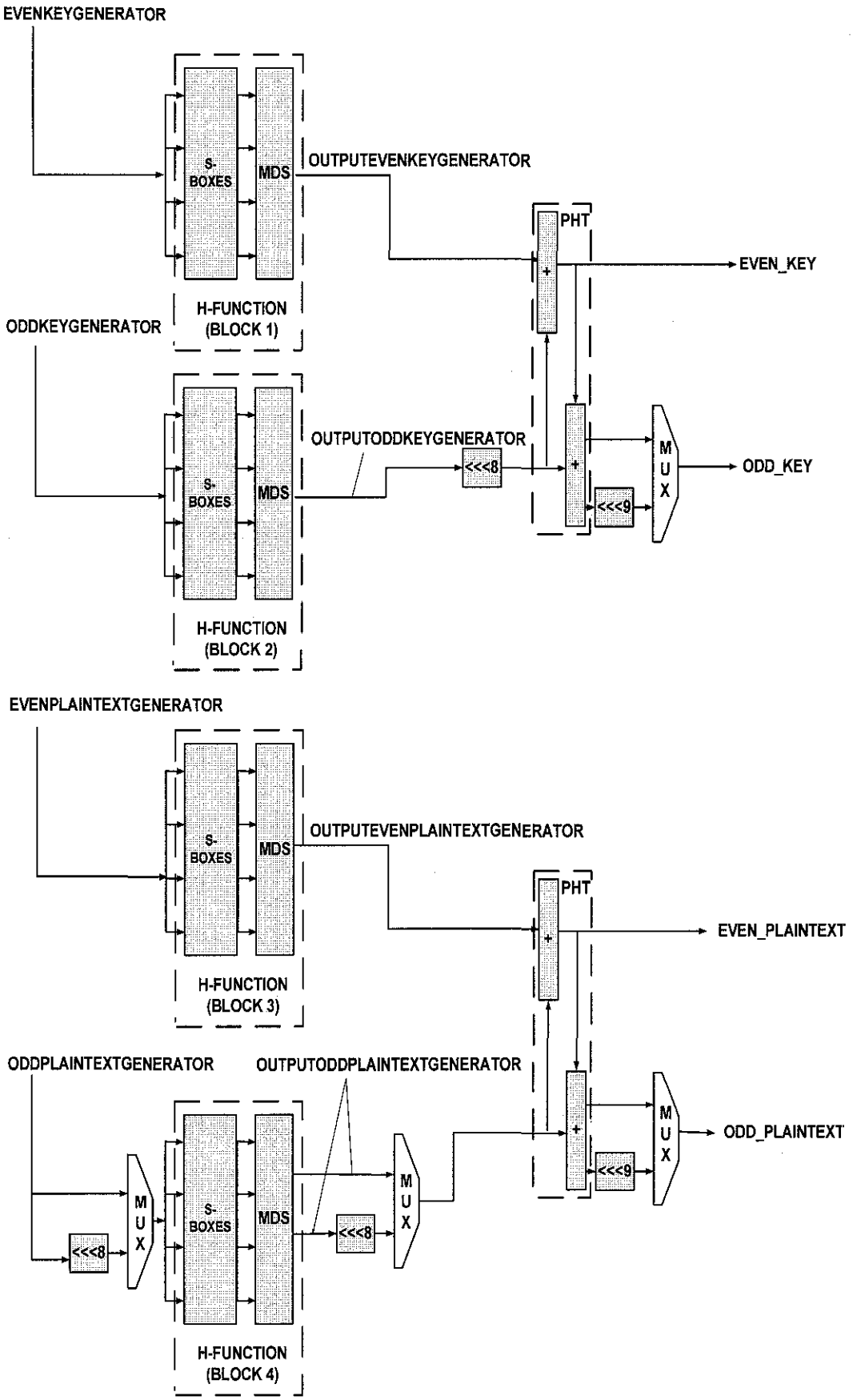


Figure 16: Generalized F-Function of Design 2

4.1.2.5 Design Overall Structure

Design 1

One of the most challenging parts is combining all the individual parts and laying them to form a complete architecture. This is shown by the diagram below. The modified F-function is the most important element and it needs to be carefully arranged with other blocks. Integrating this modified F-function to the other elements of the design resulted in the final structure shown above. The four multiplexers shown on top are used to select between the data from the clear text module used at the beginning of an encryption/decryption cycle and the data obtained after each round of encryption/decryption. **The registers at the output of the modified F function are also "pipeline" registers.** They store the values of the K-subkeys (K2r+8 and K2r+9, r ranging from 0 to 15) used in each encryption/decryption cycle. After each round the data is latched into 4 registers. This step also performs the required swap after each round. A finite state machine was implemented to control the system. The design had to be carefully implemented so that the modified F- function is highly efficient and continuously processes information.

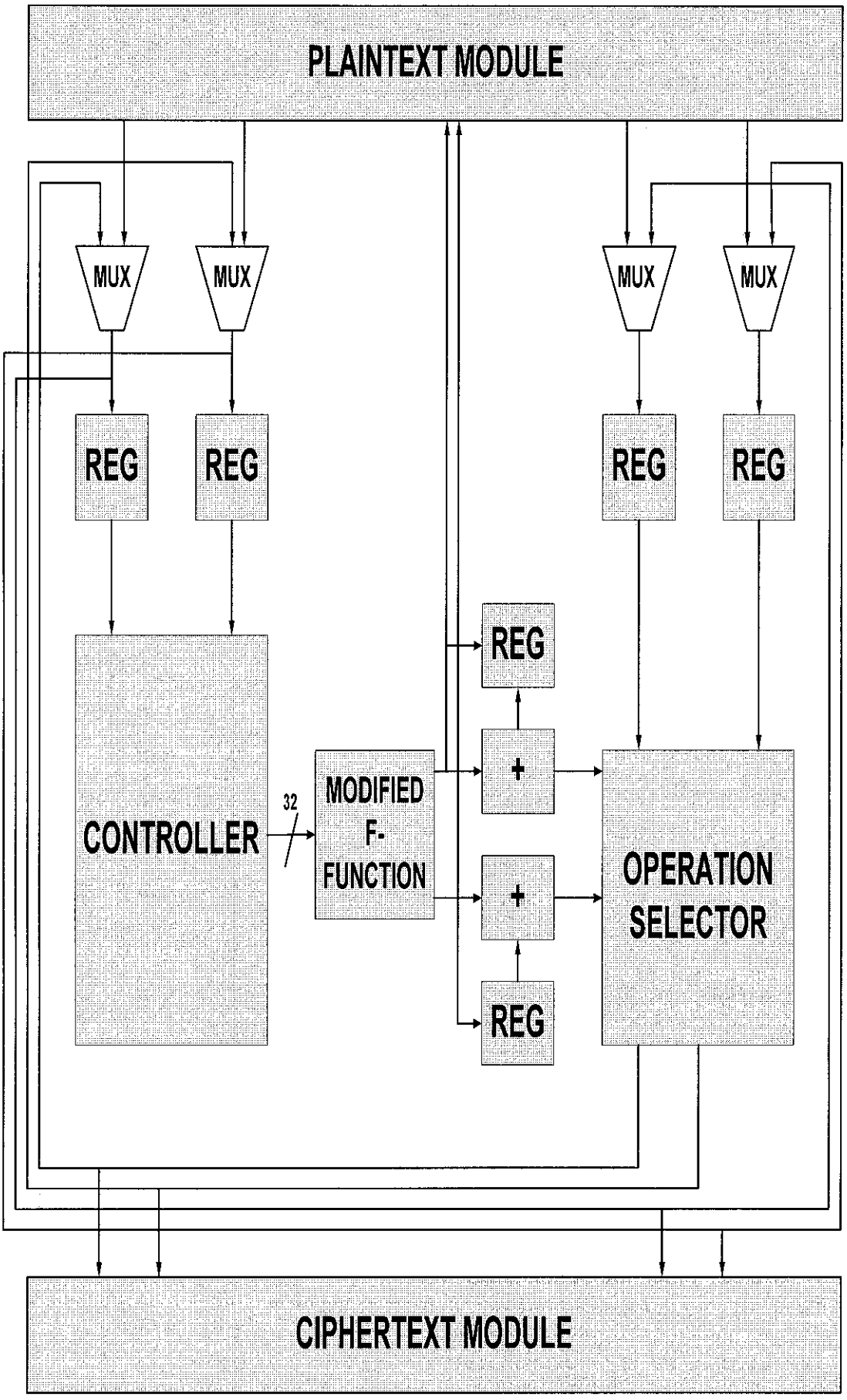


Figure 17: Structure of the Cipher of Design 1

Design 2

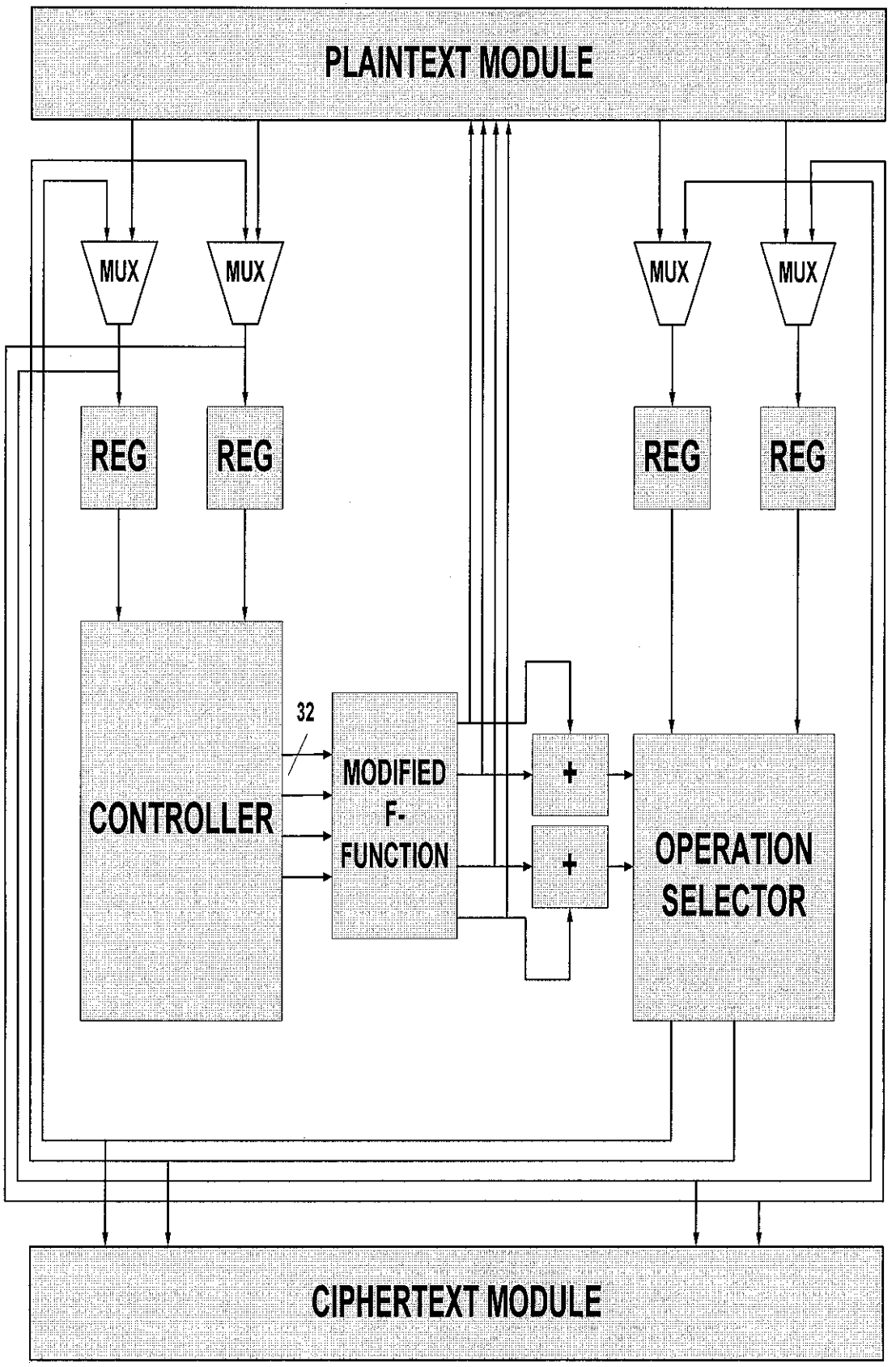


Figure 18: Structure of the Cipher of Design 2

As can be seen in the above diagram, Design 2 is almost identical to Design 1, but the difference is the modified F-Function. With this new design, we could observe that there are 4 inputs to the modified F-Function namely inputevenkeygenerator, inputoddkeygenerator, inputevenplaintextgenerator and inputoddplaintextgenerator as compared to Design 1 where there is only 1 input. **This particular new feature speeds the completion time by a factor of 4.** Besides that, we do not need any registers at the output of the modified F-Function to store any temporary values. Other than that the overall architecture is quite similar to Design 1. **It was observed that significant performance improvement could be achieved with this design.**

4.1.2.6 Controller

Design 1

One of the most challenging and tedious element is the controller. Once the architecture have been laid, it is very important to have some sort of controller mechanism to generate control signals for the proper operation of the algorithm. The cipher is controlled by a relatively complex finite state machine. While in the idle state, the controller waits for a user request and advertises its availability by raising a signal. The 'loadKey' signal makes the controller go to the 'loadKey' state in which it latches the key material from the 128-bit input port.

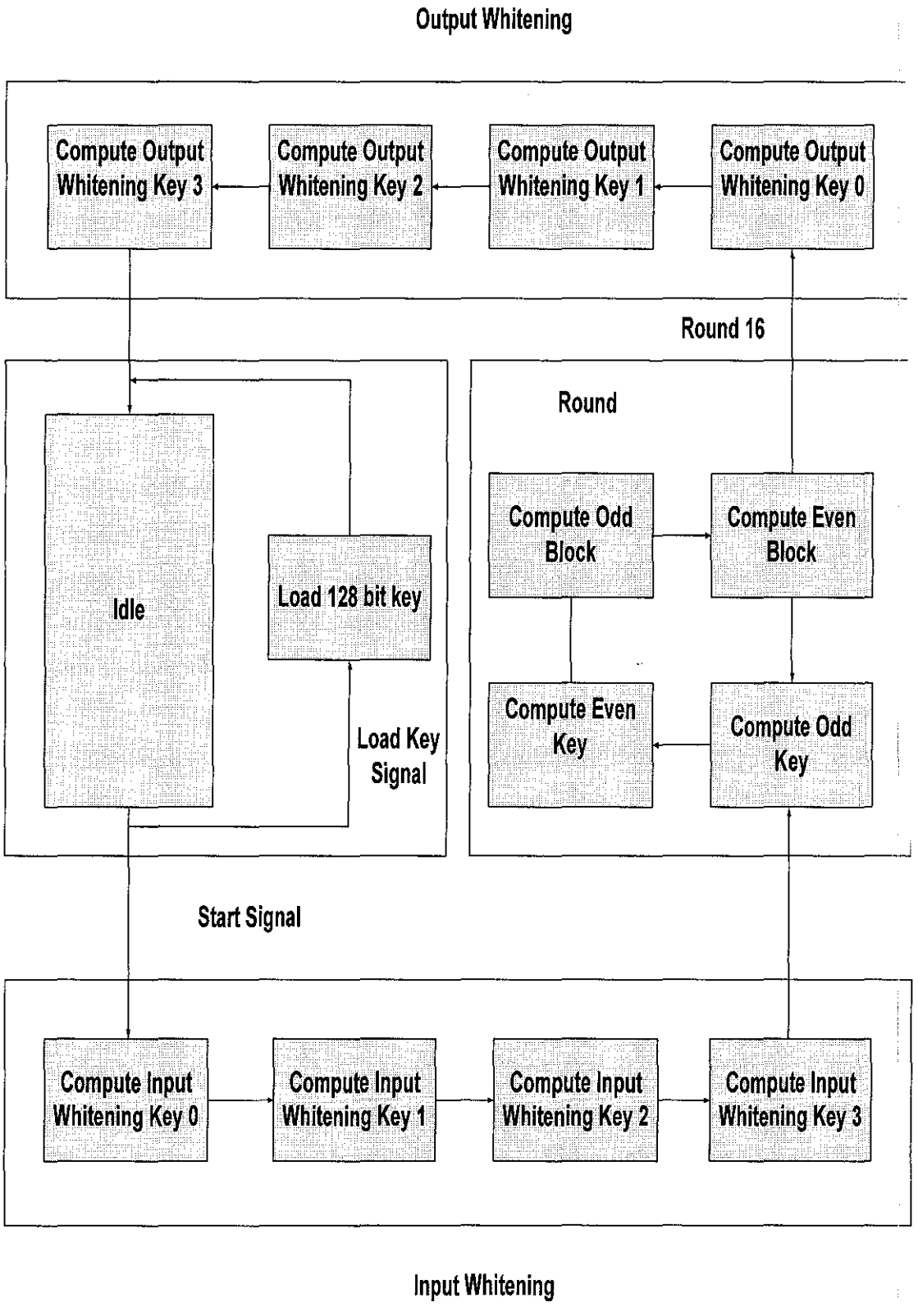


Figure 19: The Controller of Design 1

The 'start' signal puts the controller in the encryption or decryption mode. This enables a user to choose which mode the device is to be executed that is

whether in encryption or decryption mode. It is quite important to note that the controller operates almost exactly the same way when doing encryption or decryption. The first step is to compute the four input whitening sub-keys. After each pair is computed the two registers on the top left of Figure 15 are latched with the output of the input register module. After the second pair, the two other registers on the right are latched. The controller then goes through 16 rounds of encryption or decryption. These rounds consist of four states. The even and odd keys for the round are computed and latched into to registers. The F-function is then used to compute a pair of results, which are added to the content of the registers. At the end of the round the result is latched into registers. The outputs of the rounds are also swapped between the left and right side of the circuit. After the end of the 16 rounds the four output whitening sub-keys are computed and used two-by-two to whiten the output. Soon after the controller goes back to the idle state, the cipher text register is latched with the output of the process. Note that sub-keys have to be generated by setting the input of the F-function with values ranging from 0 to 39. Doing this directly in the controller would be awkward. **The controller thus uses a counter to do the job. This counter counts in the order in which the keys need to be generated (0,1,2,3,8,9,10…39,4,5,6,7 for encryption).** It can be disable so that it counts only when needed. The counting is in the reverse direction for the decryption part.

Design 2

The controller of Design 2 is almost similar to Design 1. The only difference is the controller sends control signals so that 4 elements are computed simultaneously at any one time. This could be observed in the figure below. After loading key, the controller computes the 4 input whitening keys simultaneously before proceeding to the computation of the round values. In each round, 4 values are computed namely even key, odd key, even plaintext and odd plaintext. After 16 rounds, the controller starts computing the output whitening keys. All 4 keys are computed simultaneously. Completion of this loop indicates the end of the process. Therefore, it is important to note that the controller must be designed very carefully so that proper synchronization of values takes place.

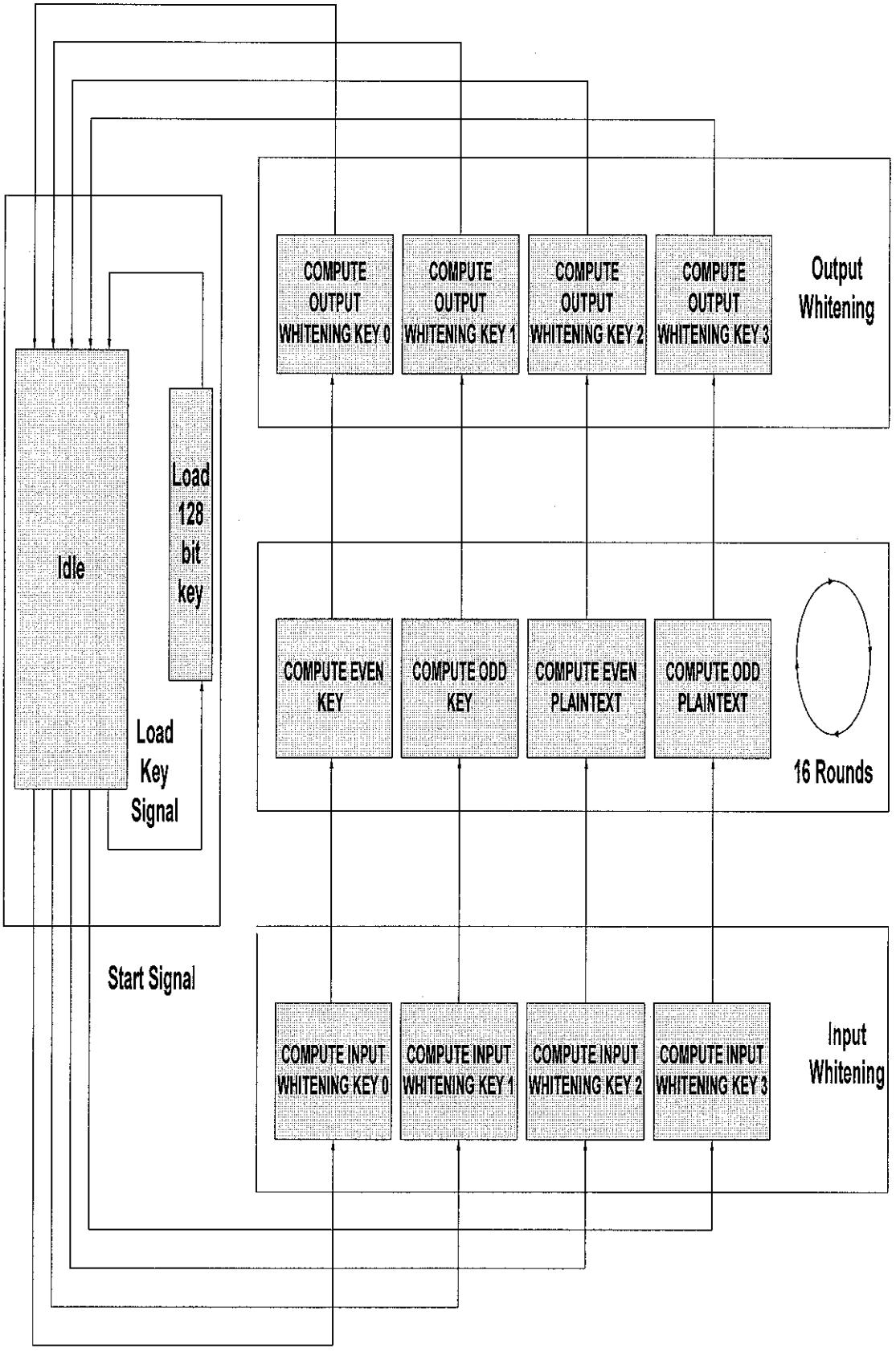


Figure 20: The Controller of Design 2

4.2 PROGRAMMING STRATEGY

I have decided to use VHDL to program the design. The simulation tools would be Aldec. The project manager would be Xilinx Webpack – ISE6.2i. With the design decisions have been laid, it is very important to strategize the programming decision. I have decided to use the top to down method. Here I break the overall design into major modules. The major modules are further broken into smaller modules until individual discrete modules are obtained. This is a very structured programming style. Besides that, breaking the design into smaller discrete units enables reuse of the modules in other bigger modules. This enables saving of time and a more efficient programming style. Furthermore discrete unit modules, enables efficient unit testing to be performed, followed by module testing, and finally integration testing. Overall hierarchy of the design flow is shown below.

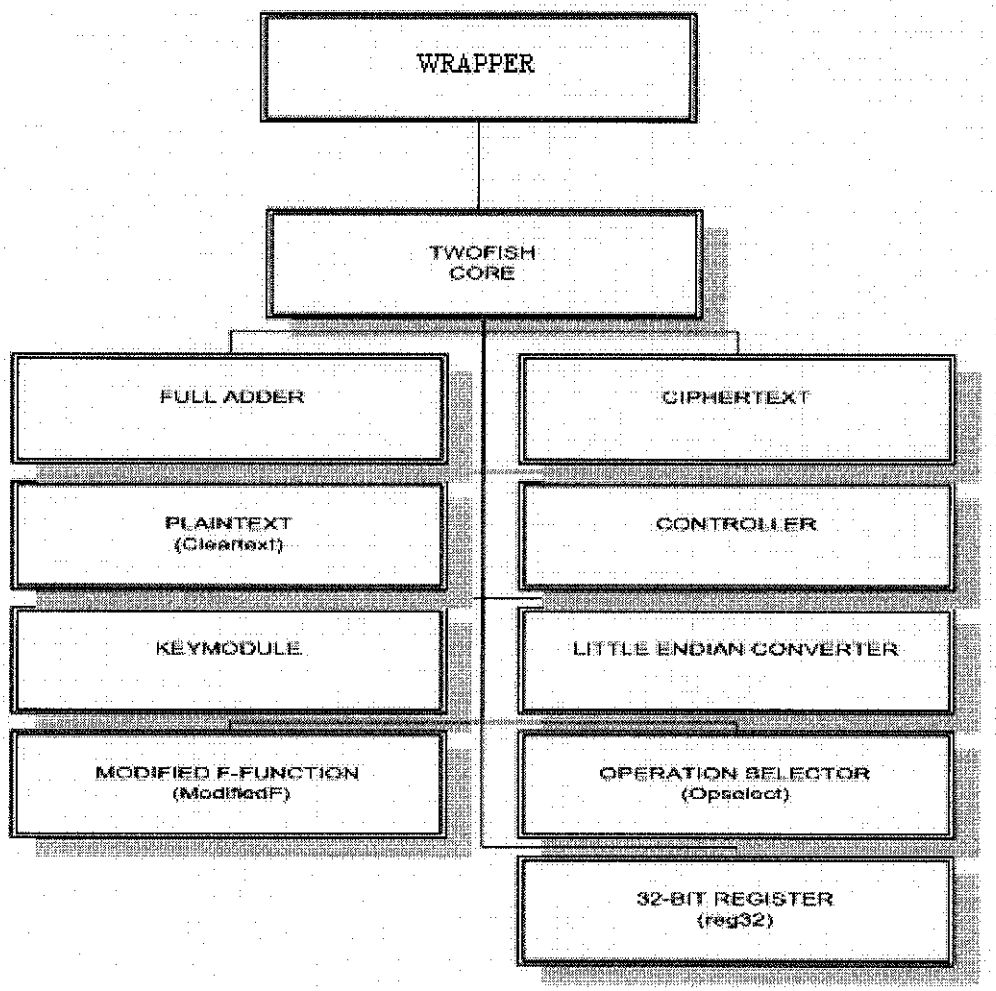


Figure 21: Hierarchy of the Design Flow

As can be seen above, the Twofish core is basically divided into 9 major sub modules. Each sub modules perform specific task. The block diagram of the various main modules is shown below. Differences between Design 1 and 2 will be highlighted where necessary.

NOTE: Only major sub modules are described.

4.2.1 Twofish Core

This component is the topmost component in the hierarchy.

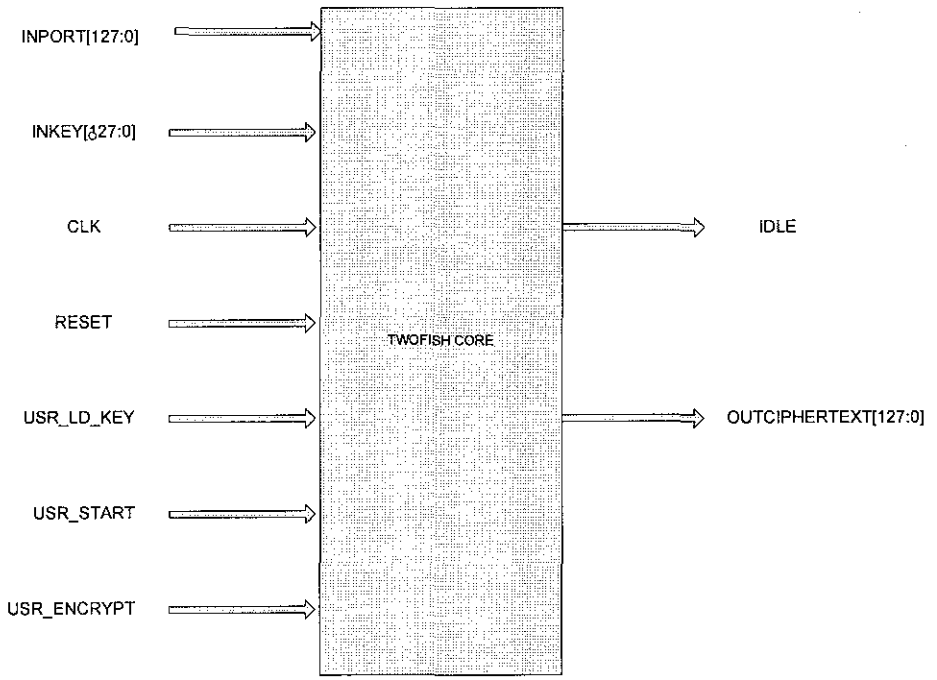


Figure 22: Block Diagram of the Twofish Core – with I/O pins

Table 1: Twofish Core

INPORT[127:0]	Accepts 128 bit of plaintext from the user.
INKEY[127:0]	Accepts 128 bit of input key from the user.
CLK	Clock
RESET	Reset
USR_LD_KEY	Loads the key.
USR_START	Starts the whole process. (Encryption/Decryption)
USR_ENCRYPT	States the process. (Encryption/Decryption)

IDLE	Shows whether the system is idle.
OUT_CIPHERTEXT [127:0]	Outputs the ciphertext.

For this component, the main operations are as follows:

Load Key

The following signals have to be high:

- INKEY= '1'
- CLK = '1'
- USR_LD_KEY = '1'
- RESET = '0'
- USR_START ='0'
- USR_ENCRYPT ='1' if encryption and '0' if decryption

Encryption

The following signals have to be high:

- INPORT= '1'
- CLK = '1'
- USR_START = '1'
- USR_ENCRYPT = '1'
- USR_LD_KEY = '1'
- RESET = '0'

Decryption

The following signals have to be high:

- INPORT= '1'
- CLK = '1'
- USR_START = '1'
- USR_ENCRYPT = '0'
- USR_LD_KEY = '1'
- RESET = '0'

The Twofish Core is basically the module that a given user can control from the top layer. By sending the appropriate signals a given user can control which process is to be executed i.e. load key, encrypt, or decrypt.

4.2.2 Full Adder

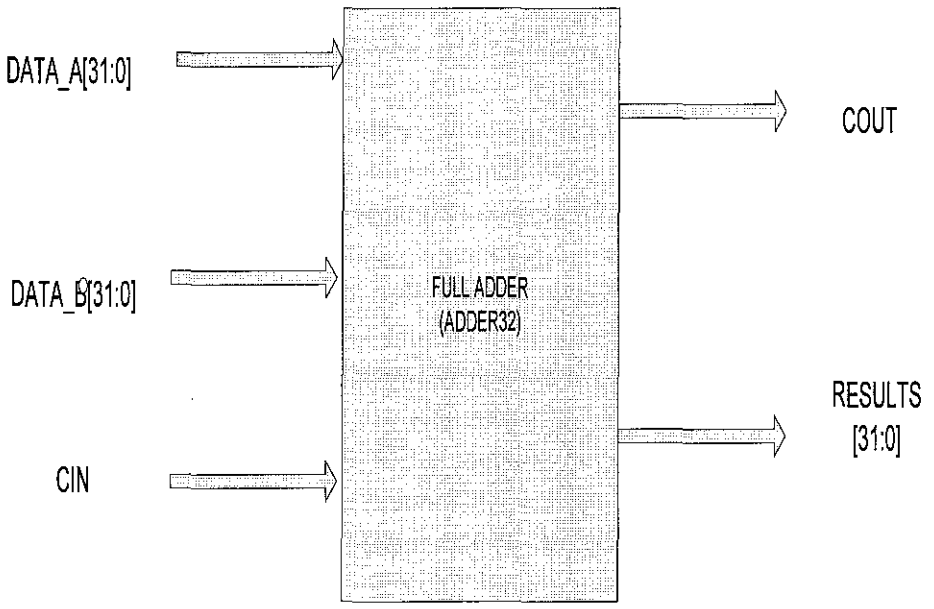


Figure 23: Block Diagram of the Full Adder

Table 2: Full Adder

DATA_A[31:0]	Accepts a 32 bit input
DATA_B[31:0]	Accepts a 32 bit input
CIN	Accepts an input carry bit
COUT	Outputs the output carry bit
RESULTS	Outputs the addition result

The full adder basically adds 2 -32bit values and outputs the appropriate results. The full adder is mostly used in PHT.

4.2.3 Ciphertext

Design 1

The part produces the encrypted data in an encryption process.

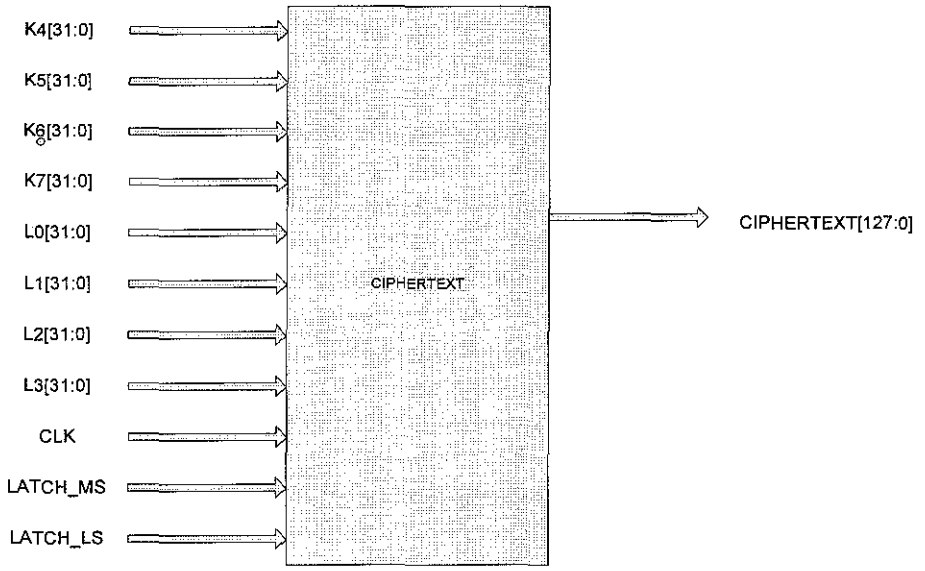


Figure 24: Block Diagram of the Ciphertext of Design 1

Table 3: Ciphertext – Design 1

K4[31:0]	Accepts K4 values.
K5[31:0]	Accepts K5 values.
K6[31:0]	Accepts K6 values.
K7[31:0]	Accepts K7 values.
L[31:0]	Accepts L0 values.
L1[31:0]	Accepts L1 values.
L2[31:0]	Accepts L2 values.
L3[31:0]	Accepts L3 values.
CLK	Clock signal
LATCH_MS	Most Significant 64 bits of output signal.
LATCH_LS	Least Significant 64 bits of output signal.
CIPHERTEXT[127:0]	Output Ciphertext.

Most Significant 64 bits of output signal

The following signals have to be HIGH or available:

K6 [31:0] = K6

K7 [31:0] = K7

L2 [31:0] = available values

L3 [31:0] = available values

CLK = '1'

LATCH_MS = '1'

Least Significant 64 bits of output signal

The following signals have to be HIGH or available:

K4 [31:0] = K4

K5 [31:0] = K5

L0 [31:0] = available values

L1 [31:0] = available values

CLK = '1'

LATCH_LS = '1'

Ciphertext

This signal is available at any time based on any immediate input values. (BUT the above constraints have to be met)

Design 2

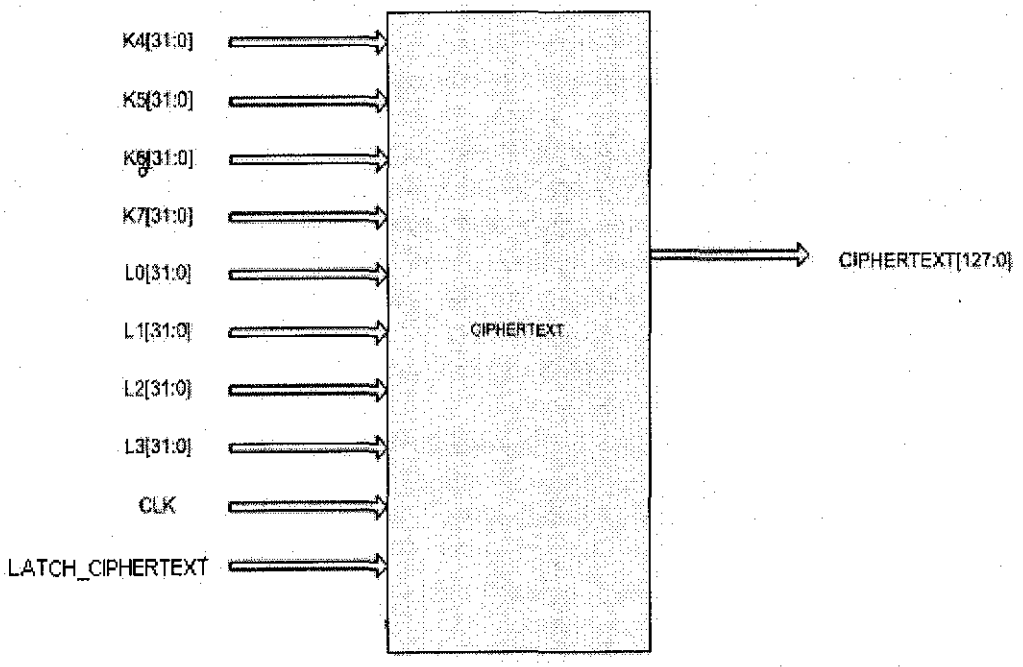


Figure 25: Block Diagram of the Ciphertext of Design 2

Table 4: Ciphertext – Design 2

K4[31:0]	Accepts K4 values.
K5[31:0]	Accepts K5 values.
K6[31:0]	Accepts K6 values.
K7[31:0]	Accepts K7 values.
L[31:0]	Accepts L0 values.
L1[31:0]	Accepts L1 values.
L2[31:0]	Accepts L2 values.
L3[31:0]	Accepts L3 values.
CLK	Clock signal
LATCH_CIPHERTEXT	Latching signal.
CIPHERTEXT[127:0]	Output Ciphertext.

Latching 128 bits of output signal

The following signals have to be HIGH or available:

K4 [31:0] = K4

K5 [31:0] = K5

K6 [31:0] = K6

K7 [31:0] = K7

L0 [31:0] = available values

L1 [31:0] = available values

L2 [31:0] = available values

L3 [31:0] = available values

CLK = '1'

LATCH_CIPHERTEXT = '1'

4.2.4 Cleartext/Plaintext

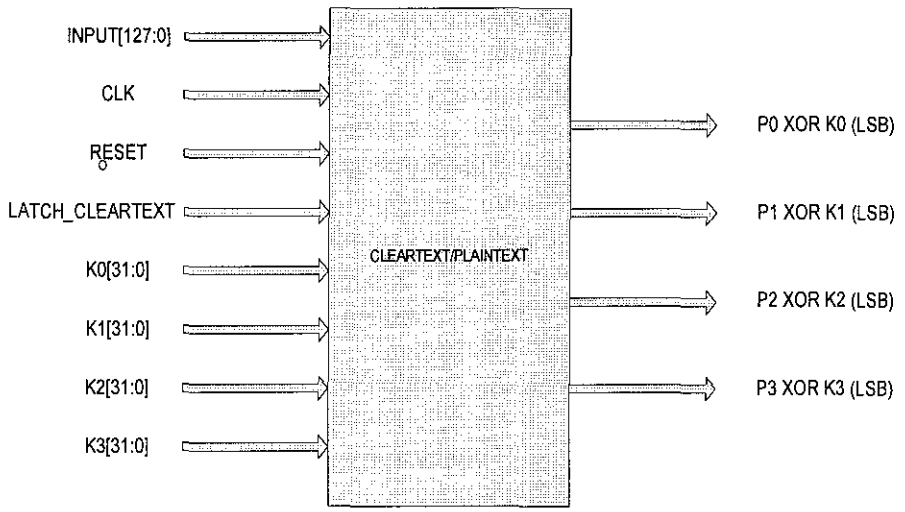


Figure 26: Block Diagram of the Cleartext/Plaintext

Table 5: Cleartext/Plaintext

INPUT[127:0]	Accepts 128 bits of input plaintext.
CLK	Clock signal.
RESET	Reset signal.
LATCH_CLEARTEXT	Signal to latch plaintext.
K0[31:0]	Key 0.
K1[31:0]	Key 1.
K2[31:0]	Key 2.
K3[31:0]	Key 3.
PO XOR KO[31:0]	PO XOR KO output.
P1 XOR K1[31:0]	P1 XOR K1 output.
P2 XOR K2[31:0]	P2 XOR K2 output.
P3 XOR K3[31:0]	P3 XOR K3 output.

P0 XOR K0 (LSB)

INPUT [31:0] = plaintext

CLK = '1'

RESET = '0'

K0 [31:0] = K0

LATCH_CLEARTEXT = '1'

P1 XOR K1

INPUT [63:32] = plaintext

CLK = '1'

RESET = '0'

K1 [31:0] = K1

LATCH_CLEARTEXT = '1'

P2 XOR K2

INPUT [95:64] = plaintext

CLK = '1'

RESET = '0'

K2 [31:0] = K2

LATCH_CLEARTEXT = '1'

P3 XOR K3

INPUT [127:96] = plaintext

CLK = '1'

RESET = '0'

K3 [31:0] = K3

LATCH_CLEARTEXT = '1'

Reset System

RESET = '1'

This part basically takes the input plaintext and XORs it with the corresponding input whitening key.

4.2.5 Controller

This is the **most complex system** of the whole process. It integrates all the components by sending the appropriate signals. By sending the wrong signals, the integrated components would not work as expected. As results a complex Finite State Machines was made to integrate the system. This is shown below.

Design 1

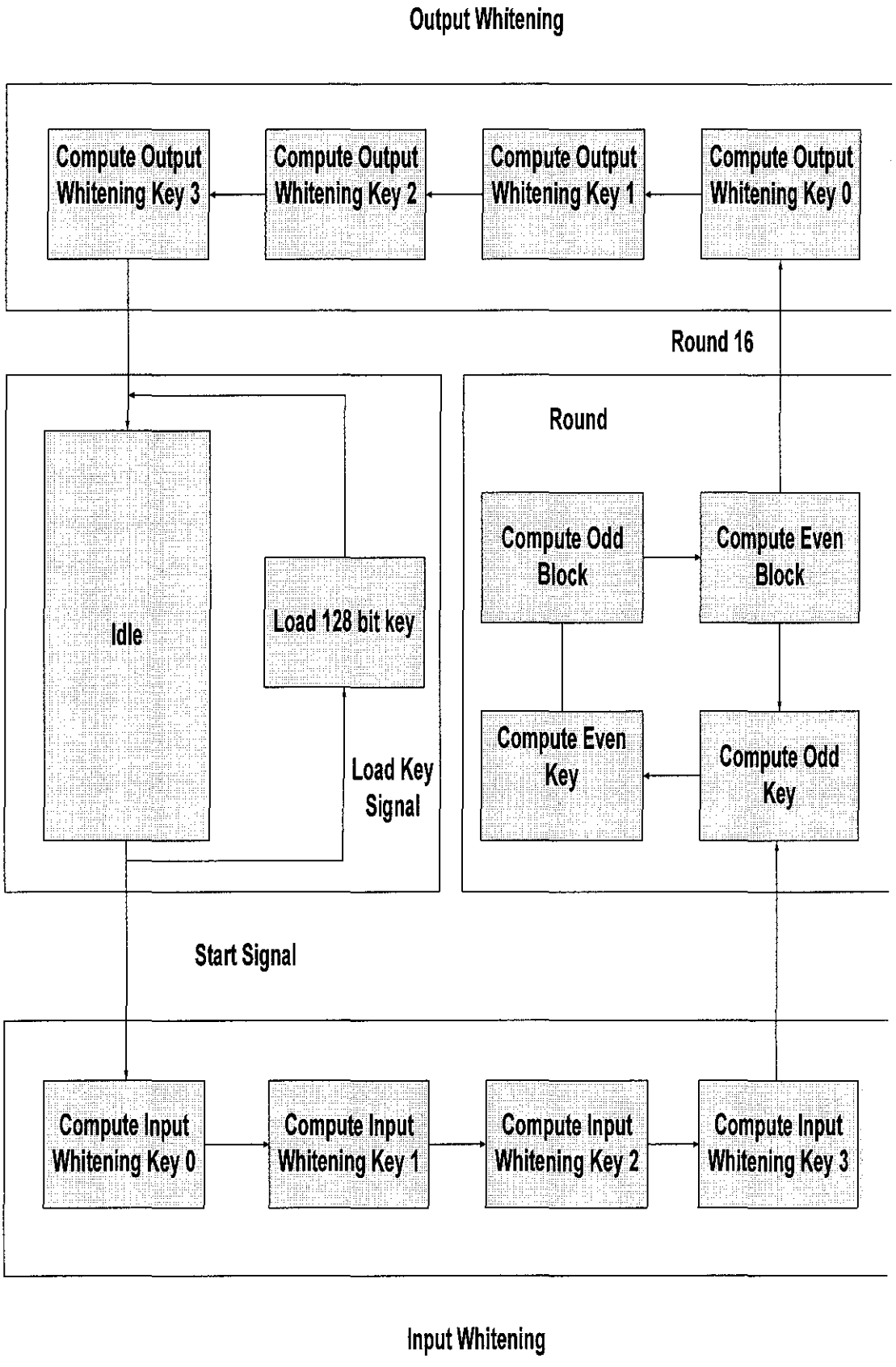


Figure 27: Finite State Machine of the Design 1

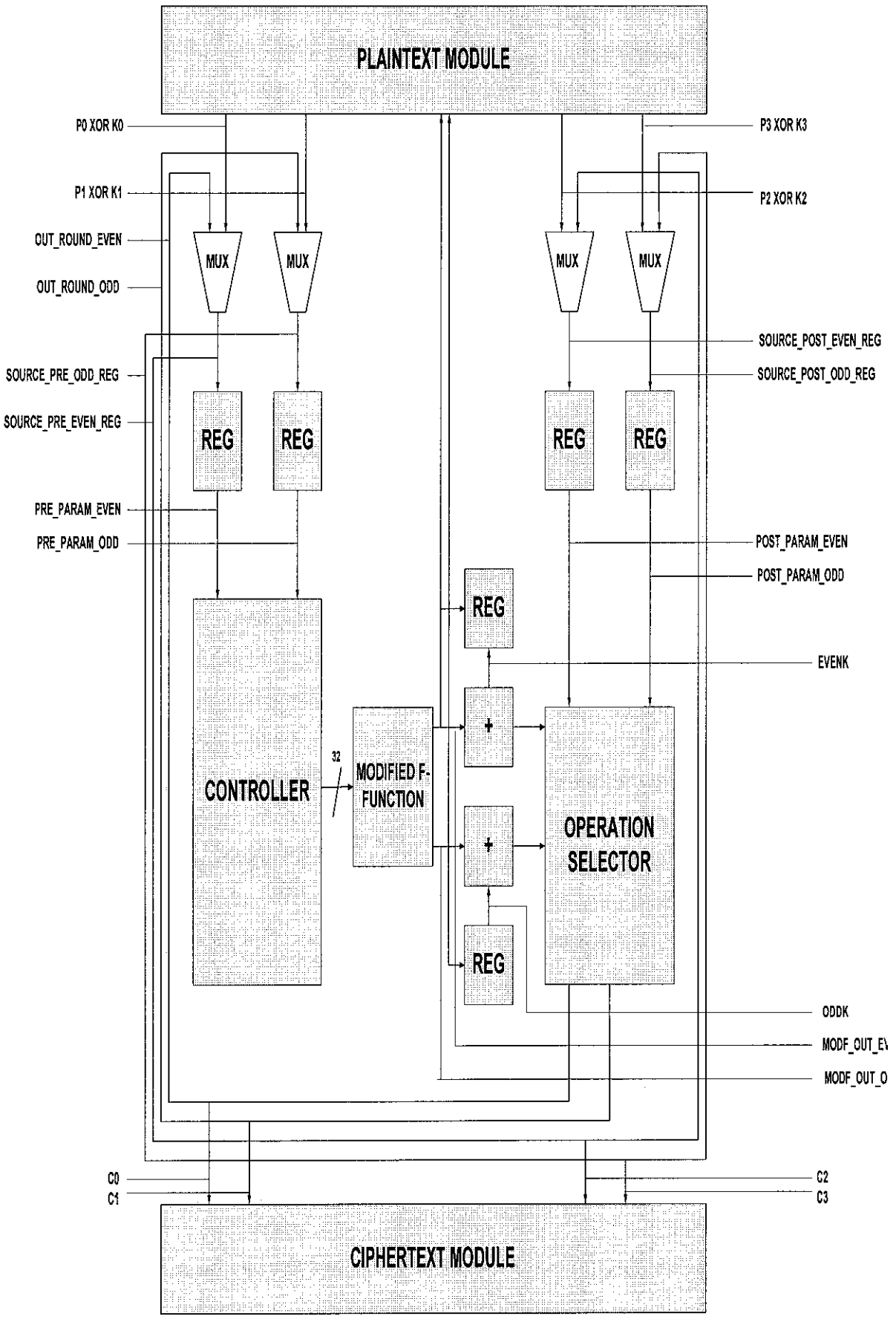


Figure 28: Overall Design with some of the Control Signals for Design 1

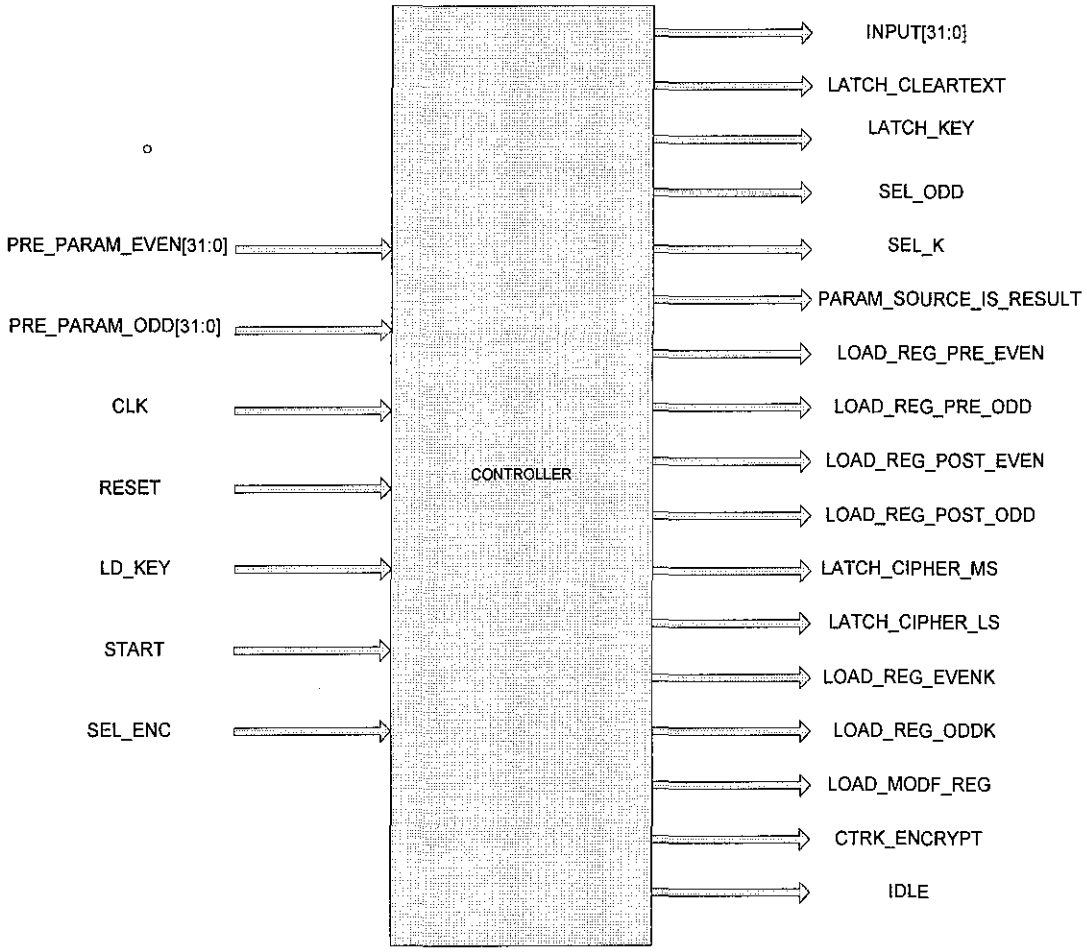


Figure 29: Block Diagram of the Controller for Design 1

Table 6: Controller – Design 1

PRE_PARAM_EVEN[31:0]	Even 32-bit input values.
PRE_PARAM_ODD[31:0]	Odd 32-bit input values.
CLK	Clock signal.
RESET	Reset signal.
LD_KEY	Load key signal.
START	Start encryption/decryption process.
SEL_ENC	Select encryption or decryption process.
INPUT_F[31:0]	Selected 32 bit input values.
LATCH_CIPHERTEXT	Signal to latch ciphertext (plaintext).
LATCH_KEY	Signal to latch key.
SEL_ODD	Signal to select odd path (odd values).
SEL_K	Signal to select key.
PARAM_SOURCE_IS_RESULT	Signal to indicate parameter source is result.

LOAD_REG_PRE_EVEN	Signal to load pre even register.
LOAD_REG_PRE_ODD	Signal to load pre odd register.
LOAD_REG_POST_EVEN	Signal to load post even register.
LOAD_REG_POST_ODD	Signal to load post odd register.
LATCH_CIPHER_MS	Signal to latch most significant 64 bit ciphertext.
LATCH_CIPHER_LS	Signal to latch least significant 64 bit ciphertext.
LOAD_REG_EVENK	Signal to load even key to register.
LOAD_REG_ODDK	Signal to load odd key to register.
LOAD_MODF_REG	Signal to load values into MODF register.
CTRL_ENCRYPT	Control encrypt signal.
IDLE	Idle status signal.

As we can see in the Finite State machine, the controller has to control both the key calculation process and also encryption/decryption process. Within these processes, some are even and some are odd. In general we can divide the whole process into 3 main processes namely as follows:

Load Key

PRE_PARAM_EVEN [31:0] = doesn't matter

PRE_PARAM_ODD [31:0] = doesn't matter

CLK = '1'

RESET = '0'

LD_KEY = '1'

START = '0'

SEL_ENC = '1' for encryption and '0' for decryption

Even Key/Plaintext

PRE_PARAM_EVEN [31:0] = even key/plaintext

PRE_PARAM_ODD [31:0] = doesn't matter

CLK = '1'

RESET = '0'

LD_KEY = '0'

START = '1'

SEL_ENC = '1' for encryption and '0' for decryption

Odd Key/Plaintext

PRE_PARAM_EVEN [31:0] = doesn't matter

PRE_PARAM_ODD [31:0] = odd key/plaintext

CLK = '1'

RESET = '0'

LD_KEY = '0'

START = '1'

SEL_ENC = '1' for encryption and '0' for decryption

From here on, it is quite surprising how the process can continue on its own. Well, there is a counter that is embedded in the controller itself. The counter counts in a fixed pattern. Once START goes to high. The pattern follows the Finite State Machine. It counts 0,1,2,3,8,9,10,……,39,4,5,6,7 for encryption. The whole counting sequence is reverse for decryption. With the counting sequence going on, appropriate states would be selected and thus appropriate control signals are generated. The necessary control signals generated for the appropriate state is shown below. **It was a very complex effort to synchronize the whole system.**

output\state	my_ idle	Load _KeyA	Compute _K0	Compute _K1	Compute _K2	Compute _K3	Compute _K4	Compute _K5	Compute _K6	Compute _K7	Compute _K2r_8	Compute _K2r_9	Compute _even	Compute _odd
input_f	Some finite Input value at input_f –KEY/DATA													
latch_key	0	1	0	0	0	0	0	0	0	0	0	0	0	0
reset_cntr	1	0	0	0	0	0	0	0	0	0	0	0	0	0
latch_cleartext	1	0	0	0	0	0	0	0	0	0	0	0	0	0
sel_odd	0	0	0	1	0	1	0	1	0	1	0	1	0	1
sel_k	0	0	1	1	1	1	1	1	1	1	1	1	0	0
param_source_i n_result_nextcc	0	0	0	0	0	0	1	1	1	1	1	1	1	1
load_reg_pre_ even_nextcc	0	0	0	1	0	0	0	0	0	0	0	0	0	1
load _reg_pre_ odd_nextcc	0	0	0	1	0	0	0	0	0	0	0	0	0	1
load_reg_post_ even_nextcc	0	0	0	0	0	1	0	0	0	0	0	0	0	1
load_reg_post_ odd_nextcc	0	0	0	0	0	1	0	0	0	0	0	0	0	1
latch_cipher_ MS_nextcc	0	0	0	0	0	0	0	0	0	1	0	0	0	0
latch_cipher_ LS_nextcc	0	0	0	0	0	0	0	1	0	0	0	0	0	0
load_reg_evenk_nextcc	0	0	0	0	0	0	0	0	0	0	0	1	0	0
load_reg_oddk_nextcc	0	0	0	0	0	0	0	0	0	0	0	1	0	0
load_modf_reg	1	1	1	1	1	1	1	1	1	1	1	1	1	1
load_encrypt	1	0	0	0	0	0	0	0	0	0	0	0	0	0
idle	1	0	0	0	0	0	0	0	0	0	0	0	0	0
cnt_enbl	-	0	1	1	1	1	1	1	1	1	1	1	0	0

Table 7: Generated Control Signals for the Corresponding States for Design 1

Design 2

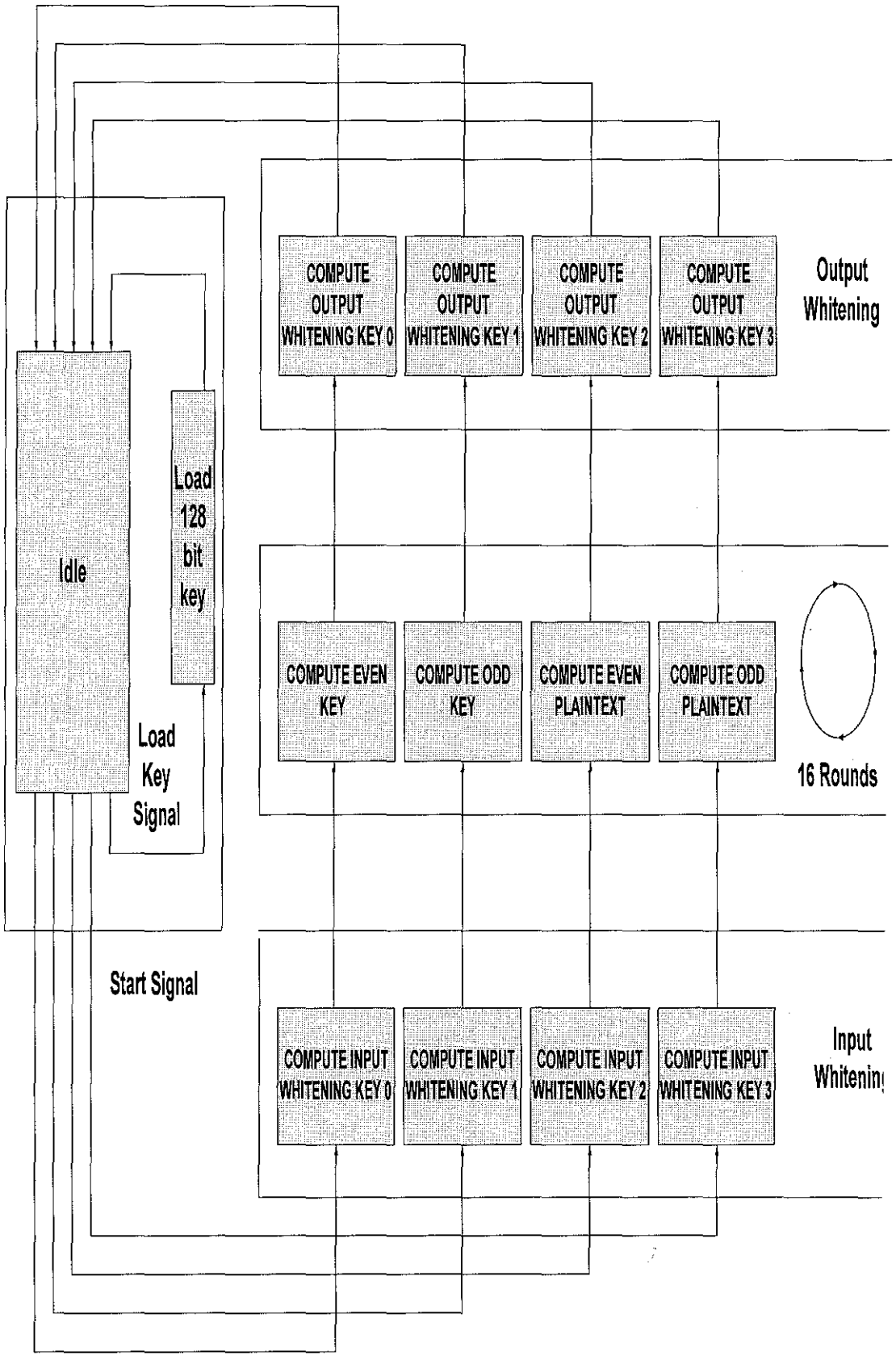


Figure 30: Finite State Machine of the Design 2

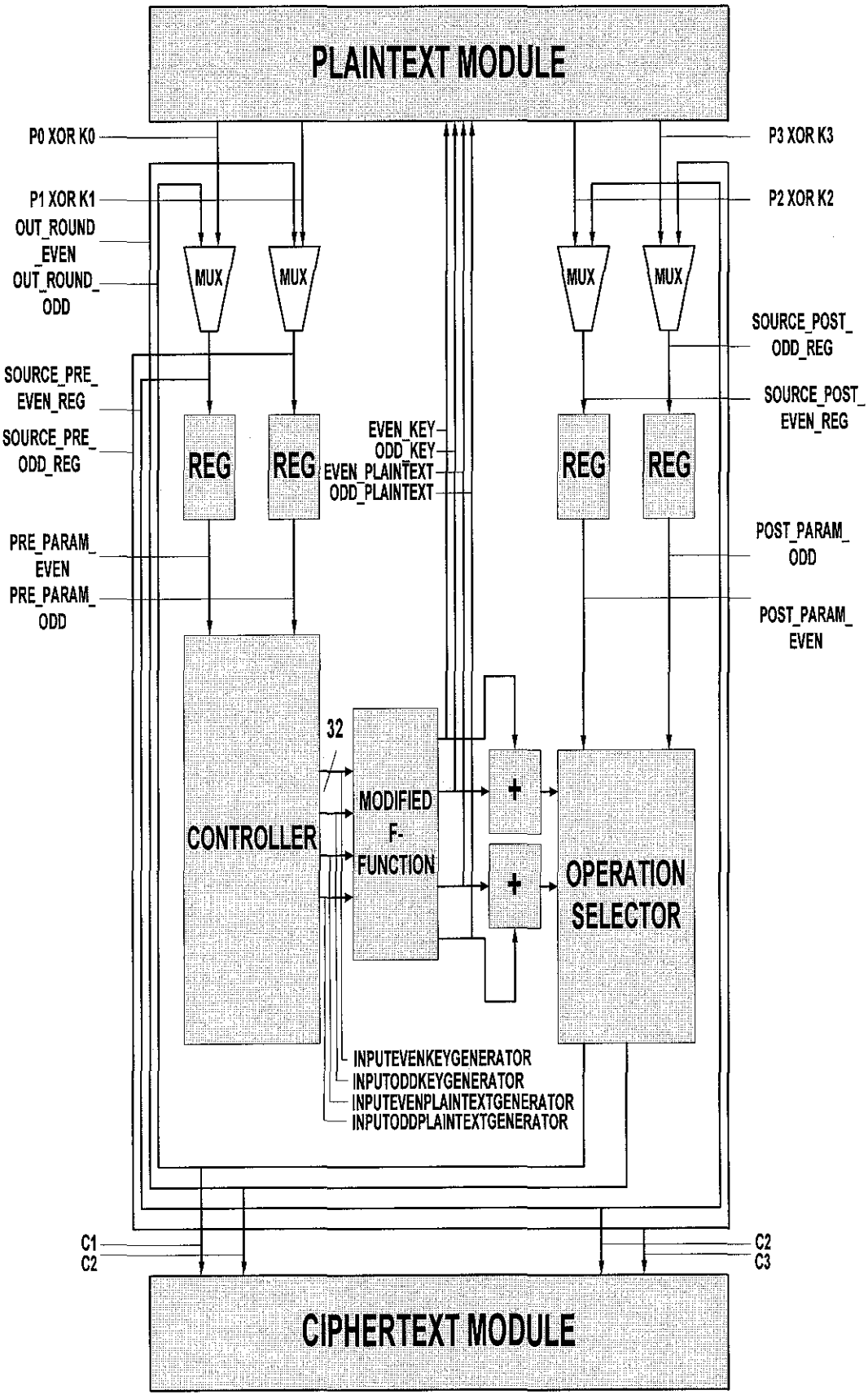


Figure 31: Overall Design with some of the Control Signals for Design 2

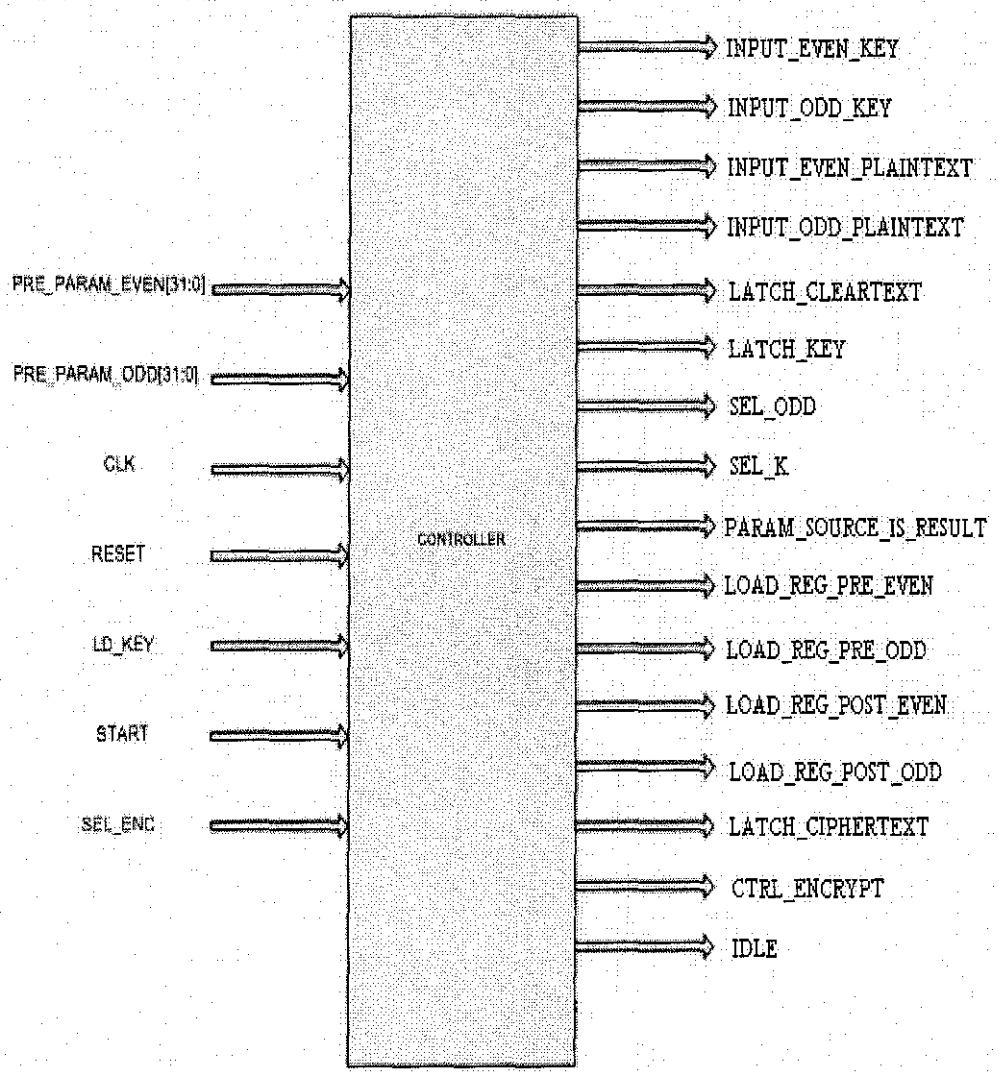


Figure 32: Block Diagram of the Controller for Design 2

Table 8: Controller – Design 2

PRE_PARAM_EVEN[31:0]	Even 32-bit input values.
PRE_PARAM_ODD[31:0]	Odd 32-bit input values.
CLK	Clock signal.
RESET	Reset signal.
LD_KEY	Load key signal.
START	Start encryption/decryption process.
SEL_ENC	Select encryption or decryption process.
INPUT_EVEN_KEY	32-bit Even key input
INPUT_ODD_KEY	32-bit Odd key input

INPUT_EVEN_PLAINTEXT	32-bit Even plaintext input
INPUT_ODD_PLAINTEXT	32-bit Odd plaintext input
LATCH_CLEARTEXT	Signal to latch cleartext
LATCH_KEY	Signal to latch key.
SEL_ODD	Signal to select odd path (odd values).
SEL_K	Signal to select key.
PARAM_SOURCE_IS_RESULT	Signal to indicate parameter source is result.
LOAD_REG_PRE_EVEN	Signal to load pre even register.
LOAD_REG_PRE_ODD	Signal to load pre odd register.
LOAD_REG_POST_EVEN	Signal to load post even register.
LOAD_REG_POST_ODD	Signal to load post odd register.
LATCH_CIPHERTEXT	Signal to latch ciphertext (plaintext).
LATCH_CIPHERTEXT	Signal to latch ciphertext.
CTRL_ENCRYPT	Control encrypt signal.
IDLE	Idle status signal.

Once again as we can see in the Finite State machine, the controller has to control both the key calculation process and also encryption/decryption process. Within these processes, some are even and some are odd. In general we can divide the whole process into 3 main processes namely as follows. These processes for Design 2 looks similar as Design 1 but there are some differences indeed internally:

Load Key

PRE_PARAM_EVEN [31:0] = doesn't matter

PRE_PARAM_ODD [31:0] = doesn't matter

CLK = '1'

RESET = '0'

LD_KEY = '1'

START = '0'

SEL_ENC = '1' for encryption and '0' for decryption

Even Key/Plaintext

PRE_PARAM_EVEN [31:0] = even key/plaintext

PRE_PARAM_ODD [31:0] = doesn't matter

CLK = '1'

RESET = '0'

LD_KEY = '0'

START = '1'

SEL_ENC = '1' for encryption and '0' for decryption

Odd Key/Plaintext

PRE_PARAM_EVEN [31:0] = doesn't matter

PRE_PARAM_ODD [31:0] = odd key/plaintext

CLK = '1'

RESET = '0'

LD_KEY = '0'

START = '1'

SEL_ENC = '1' for encryption and '0' for decryption

From here on, it is quite surprising how the process can continue on its own. Well, there is a counter that is embedded in the controller itself. The counter counts in a fixed pattern. Once START goes to high. The pattern follows the Finite State Machine. It counts 0,1,2,3,8,9,10,……,39,4,5,6,7 for encryption. Remember for Design 2, the counter supplies values in blocks of 4 namely 0, 1 2 and 3, the next block would be 8, 9, 10 and 11, and so on. The whole counting sequence is reverse for decryption. With the counting sequence going on, appropriate states would be selected and thus appropriate control signals are generated. The necessary control signals generated for the appropriate state is shown below. **It was a very complex effort to synchronize the whole system.**

4.2.6 Keymodule

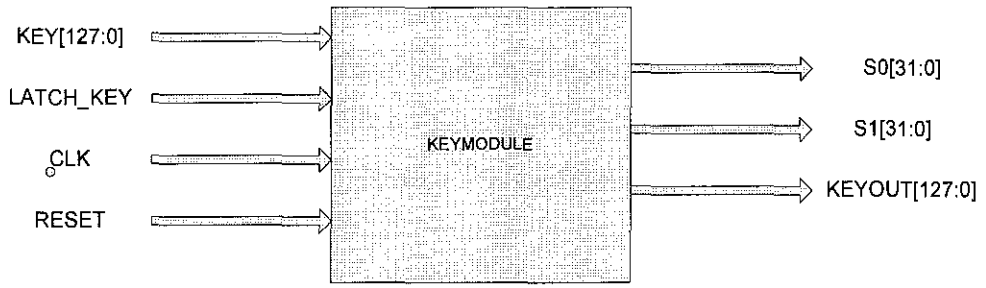


Figure 33: Block Diagram of the Keymodule

This module basically latches the input key. Then it outputs 2 derived values (S0 and S1) using RS matrix. It also outputs the latched input key values.

Table 9: Keymodule

KEY[127:0]	128 bit input key.
LATCH_KEY	Signal to latch input key.
CLK	Clock signal.
RESET	Reset signal.
S0[31:0]	Derived values using RS matrix (S0)
S1[31:0]	Derived values using RS matrix (S1)
KEYOUT[127:0]	Output latched key.

4.2.7 Modified F_Function

Design 1

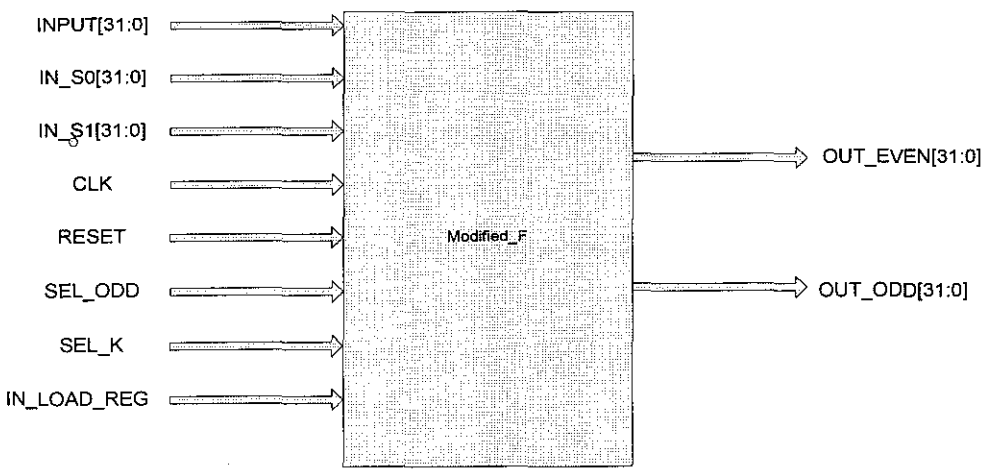


Figure 34: Block Diagram of the Modified_F of Design 1

Modified_F is the most important component. It basically computes the even and odd part of the Key and encrypted data. (blocks).

Table 10: Modified_F – Design 1

INPUT[31:0]	Input values to be computed – Key / data
IN_S0[31:0]	Derived Values – S0
IN_S1[31:0]	Derived Values –S1
CLK	Clock signal
RESET	Reset signal
SEL_ODD	Odd signal.
SEL_K	Key signal.
IN_LOAD_REG	Signal to load odd register.
OUT_EVEN[31:0]	Even output values.
OUT_ODD[31:0]	Odd output values.

Even Key

INPUT [31:0] = Even Key -32 -bits

IN_S0 [31:0] = Derived values – S0 -32 bits

IN_S1 [31:0] = Derived values – S1 -32-bits

CLK = '1'

RESET = '0'

SEL_ODD = '0'

SEL_K ='1'

IN_LOAD_REG ='0'

Odd Key

INPUT [31:0] = Even Key -32 -bits

IN_S0 [31:0] = Derived values – S0 -32 bits

IN_S1 [31:0] = Derived values – S1 -32-bits

CLK = '1'

RESET = '0'

SEL_ODD = '1'

SEL_K ='1'

IN_LOAD_REG ='1'

Even Encrypted Data Blocks

INPUT [31:0] = Even Key -32 -bits

IN_S0 [31:0] = Derived values – S0 -32 bits

IN_S1 [31:0] = Derived values – S1 -32-bits

CLK = '1'

RESET = '0'

SEL_ODD = '0'

SEL_K ='0'

IN_LOAD_REG ='0'

Odd Encrypted Data Blocks

INPUT [31:0] = Even Key -32 -bits

IN_S0 [31:0] = Derived values – S0 -32 bits

IN_S1 [31:0] = Derived values – S1 -32-bits

CLK = '1'

RESET = '0'

SEL_ODD = '1'

SEL_K ='0'

IN_LOAD_REG ='1'

Design 2

Modified_F is the most important component for Design 2 also. For Design 2, there are 4 components that make up the Modified F-Function namely:-

- EVENKEYGENERATOR
- ODDKEYGENERATOR
- EVENPLAINTEXTGENERATOR
- ODD PLAINTEXTGENERATOR

EVENKEYGENERATOR

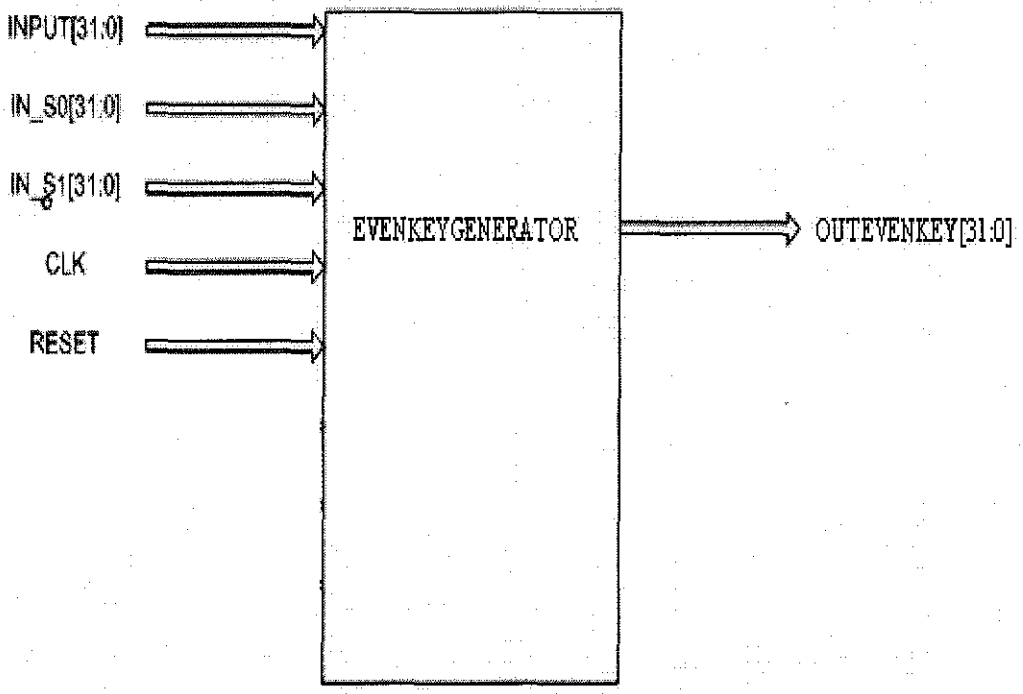


Figure 35: Block Diagram of Evenkeygenerator

Table 11: Evenkeygenerator

INPUT[31:0]	Input value – Even Key
IN_S0[31:0]	Derived Values – S0
IN_S1[31:0]	Derived Values –S1
CLK	Clock signal
RESET	Reset signal
OUTEVENKEY	Output value – Even Key

Even Key

INPUT [31:0] = Even Key –Input - 32 –bits

IN_S0 [31:0] = Derived values – S0 -32 bits

IN_S1 [31:0] = Derived values – S1 -32-bits

CLK = '1'

RESET = '0'

OUTEVENKEY = Even Key –Output - 32 –bits

ODDKEYGENERATOR

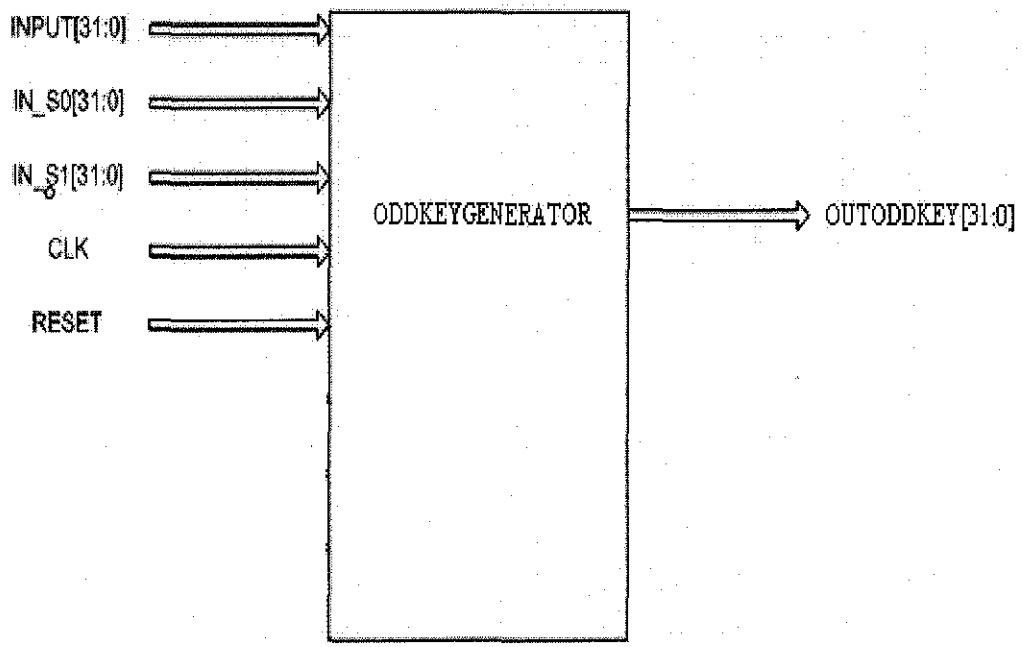


Figure 36: Block Diagram of Oddkeygenerator

Table 12: Oddkeygenerator

INPUT[31:0]	Input value – Odd Key
IN_S0[31:0]	Derived Values – S0
IN_S1[31:0]	Derived Values –S1
CLK	Clock signal
RESET	Reset signal
OUTODDKEY	Output value – Odd Key

Odd Key

INPUT [31:0] = Odd Key –Input - 32 –bits

IN_S0 [31:0] = Derived values – S0 -32 bits

IN_S1 [31:0] = Derived values – S1 -32-bits

CLK = '1'

RESET = '0'

OUTODDKEY = Odd Key –Output - 32 –bits

EVENPLAINTEXTGENERATOR

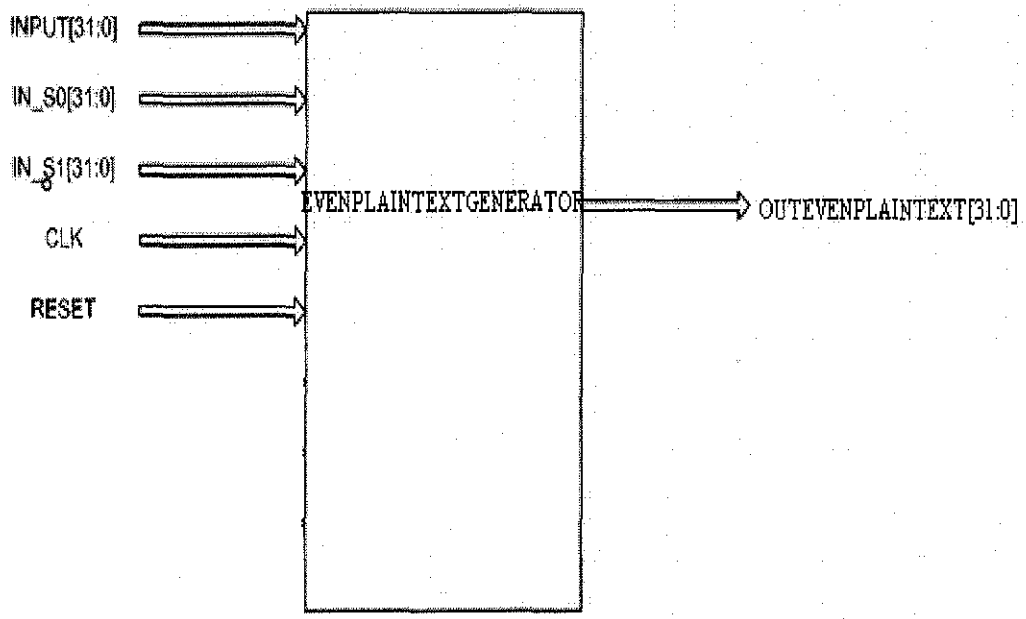


Figure 37: Block Diagram of Evenplaintextgenerator

Table 13: Evenplaintextgenerator

INPUT[31:0]	Input value – Even Plaintext/Even Key
IN_S0[31:0]	Derived Values – S0
IN_S1[31:0]	Derived Values –S1
CLK	Clock signal
RESET	Reset signal
OUTEVENPLAINTEXT	Output value – Even Plaintext/Even Key

Even Plaintext / Even Key

INPUT [31:0] = Even Plaintext –Input - 32 –bits

IN_S0 [31:0] = Derived values – S0 -32 bits

IN_S1 [31:0] = Derived values – S1 -32-bits

CLK = '1'

RESET = '0'

OUTEVENPLAINTEXT = Even Plaintext / Even Key –Output - 32 –bits

ODDPLAINTEXTGENERATOR

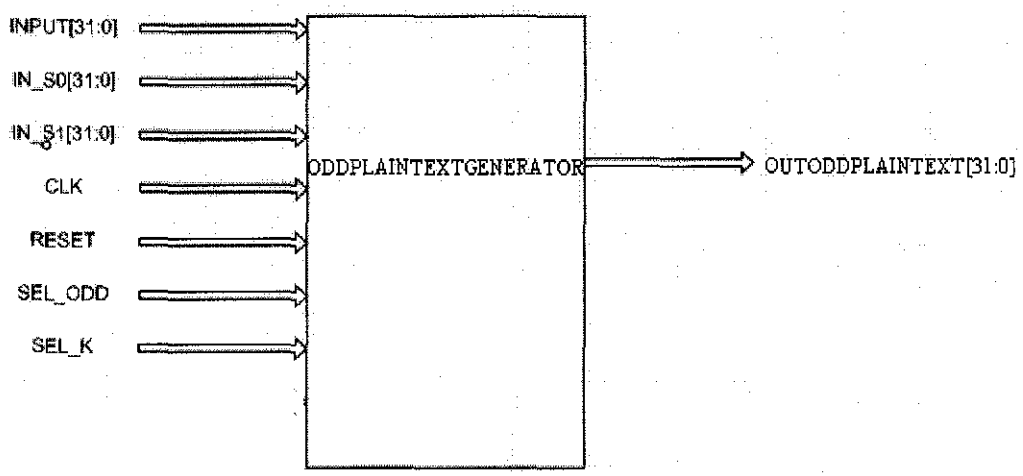


Figure 38: Block Diagram of Oddplaintextgenerator

Table 14: Oddplaintextgenerator

INPUT[31:0]	Input value – Odd Plaintext/ Odd Key
IN_S0[31:0]	Derived Values – S0
IN_S1[31:0]	Derived Values –S1
CLK	Clock signal
SEL_ODD	Selecting Odd Signal
SEL_K	Selecting Key
RESET	Reset signal
OUTODDPLAINTEXT	Output value – Odd Plaintext/Odd Key

Odd Plaintext

INPUT [31:0] = Odd Plaintext –Input - 32 –bits

IN_S0 [31:0] = Derived values – S0 -32 bits

IN_S1 [31:0] = Derived values – S1 -32-bits

CLK = '1'

SEL_ODD = '1'

SEL_K = '0'

RESET = '0'

OUTODDPLAINTEXT = Odd Plaintext –Output - 32 –bits

Odd Key

INPUT [31:0] = Odd Key –Input - 32 –bits

IN_S0 [31:0] = Derived values – S0 -32 bits

IN_S1 [31:0] = Derived values – S1 -32-bits

CLK = '1'

SEL_ODD = '1'

SEL_K = '1'

RESET = '0'

OUTODDPLAINTEXT = Odd Key –Output - 32 –bits

NOTE: Evenplaintextgenerator and Oddplaintextgenerator can calculate both plaintext and key values.

4.2.8 Opselect

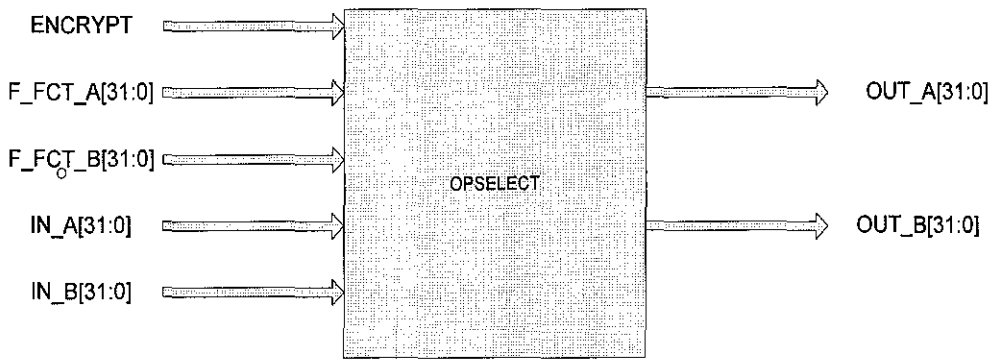


Figure 39: Block Diagram of the Opselect

Table 15: Opselect

ENCRYPT	Signal to indicate encryption or decryption process.
F_FCT_A[31:0]	Even output from Function F – 32 bits
F_FCT_B[31:0]	Odd output from Function F - 32 bits
IN_A[31:0]	Input from post_param _even – 32 bits
IN_B[31:0]	Input from post param _odd – 32 bits
OUT_A[31:0]	Even output -32 bits
OUT_B[31:0]	Odd output – 32 bits

Encryption

ENCRYPT = '1'

F_FCT_A [31:0] = Even output from Function F

F_FCT_B [31:0] = Odd output from Function F

IN_A [31:0] = Input from post_param_even

IN_B [31:0] = Input from post_param_odd

Decryption

ENCRYPT = '0'

F_FCT_A [31:0] = Even output from Function F

F_FCT_B [31:0] = Odd output from Function F

IN_A [31:0] = Input from post_param_even

IN_B [31:0] = Input from post_param_odd

This part basically performs the necessary rotation based on the chosen process – either encryption or decryption.

4.2.9 32 bit Register

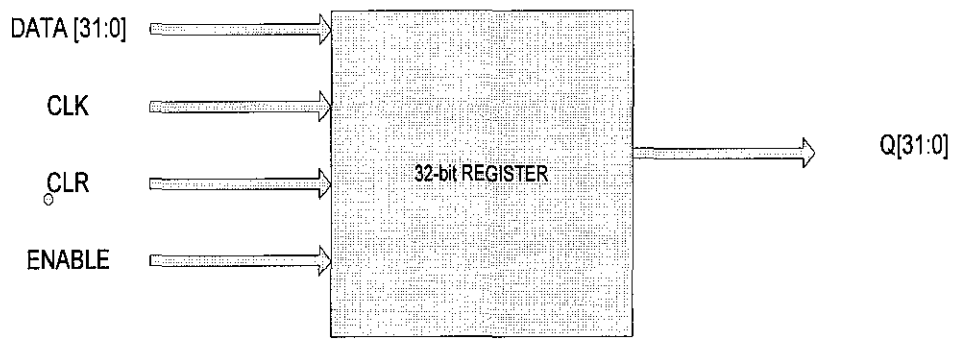


Figure 40: Block Diagram of the 32-bit Register

Table 16: 32-bit Register

DATA[31:0]	Accepts a 32 bit value
CLK	Clock signal
CLR	Clear signal
ENABLE	Enable signal
Q[31:0]	Output value that have been latched -32 bits

This component basically latches 32 bit values into the register.

4.2.10 Wrapper

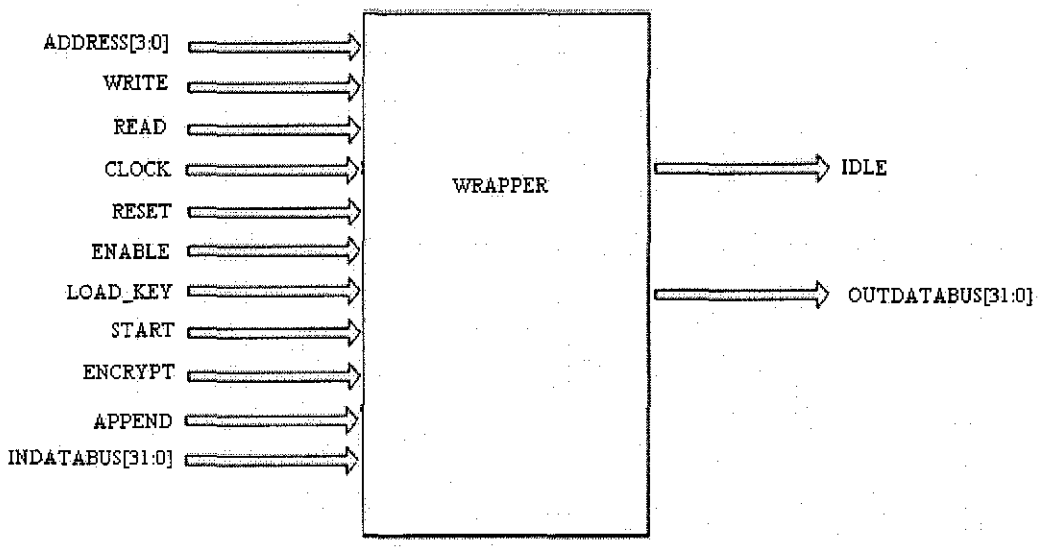


Figure 41: Wrapper for Both Design 1 and Design 2

Due to the limitation of the I/O ports of the Spartan 2 FPGAXC2S200 which has only 140 I/O ports, a wrapper is needed to download the design into the FPGA. To have a 128 individual pins for plaintext, key and ciphertext each would not be enough for the FPGA. We have to consider that we need I/O pins for control signals too. Therefore, I have decided to build a RAM for the purpose of storing, plaintext, key and ciphertext values. The RAM that I built is shown below.

Table 17: RAM for Storing Data Blocks

ADDRESS	DATA BLOCK (32 bits each)
1011	CIPHERTEXT 3
1010	CIPHERTEXT 2
1001	CIPHERTEXT 1
1000	CIPHERTEXT 0
0111	KEY3
0110	KEY 2
0101	KEY 1
0100	KEY 0

0011	PLAINTEXT 3
0010	PLAINTEXT 2
0001	PLAINTEXT 1
0000	PLAINTEXT O

To input or output any data blocks, the necessary address and control signals have to be specified. This wrapper could be used for both Design 1 and Design 2.

Table 18: Wrapper

ADDRESS[3:0]	Signal to indicate the address location in the RAM
WRITE	Write signal.
READ	Read signal
CLOCK	Clock signal
RESET	Reset signal
ENABLE	Enable signal
LOAD_KEY	Load key signal
START	Signal to indicate the start of a process (encryption/decryption)
ENCRYPT	Signal to indicate encryption/decryption
APPEND	Signal to append individual data blocks that have been extracted
INDATABUS[31:0]	32-bit input databus
IDLE	Signal to indicate idle
OUTDATABUS[31:0]	32-bit ouput databus

Writing Data into the Data Block

ADDRESS[3:0] = Any selected RAM address

WRITE = '1'

READ = '0'

RESET = '0'

ENABLE = '1'

INDATABUS[31:0] = Input data to be written

Reading Data from the Data Block

ADDRESS[3:0] = Any selected RAM address

WRITE = '0'

READ = '1'

RESET = '0'

ENABLE = '0'

INDATABUS[31:0] = Input data to be written

Appending Data

Enable = '1'

Append = '1'

NOTE : Appending data means joining 4 data blocks of 32 bits each to obtain a larger data block of 128 bits. It appends everything. (Plaintext, Key and Ciphertext)

Example: **KEY = KEY 3 & KEY2 & KEY1 & KEY0 (128 bits)**

Load Key

ENABLE = '1'

LOAD_KEY = '1'

START = '0'

RESET = '0'

READ = '0'

WRITE = '0'

APPEND = '1;

Encryption

ENABLE = '1'

LOAD_KEY = '0'

START = '1'

ENCRYPT = '1'

RESET = '0'

READ = '0'

WRITE = '0'

APPEND = '1;

Decryption

ENABLE = '1'

LOAD_KEY = '0'

START = '1'

ENCRYPT = '0'

RESET = '0'

READ = '0'

WRITE = '0'

APPEND = '1;

CHAPTER 5: RESULTS

5.1 SIMULATION

Two simulations each for Design 1 and Design 2 will be performed just to justify the validity of the results obtained. After implementing the Twofish Encryption Algorithm using VHDL, the system is ready for simulation. The system that was built is capable of performing both encryption and decryption. Official test vector given by the author during simulation to verify that the system is working in a consistent manner. The whole process is explained below:-

5.2 TEST VECTOR 1 – DESIGN 1

The official test vector from the authors is as follows:-

KEY : 00000000000000000000000000000000
PLAINTEXT : 00000000000000000000000000000000
CIPHERTEXT: 9F589F5CF6122C32B6BFEC2F2AE8C35A (expected)

With this test vector, the simulation would be carried out for both encryption and decryption.

5.2.1Encryption

The following are the main steps that need to be followed, before beginning the process.

- Load Key
- Start Encryption

5.2.1.1 Load Key

Before we start this process, the following signals need to be set first.

- INKEY= 00000000000000000000000000000000
- CLK = '1'
- USR_LD_KEY = '1'
- RESET = '0'
- USR_START ='0'
- USR_ENCRYPT ='1'

The clock speed that is used in the simulation is 10MHz.

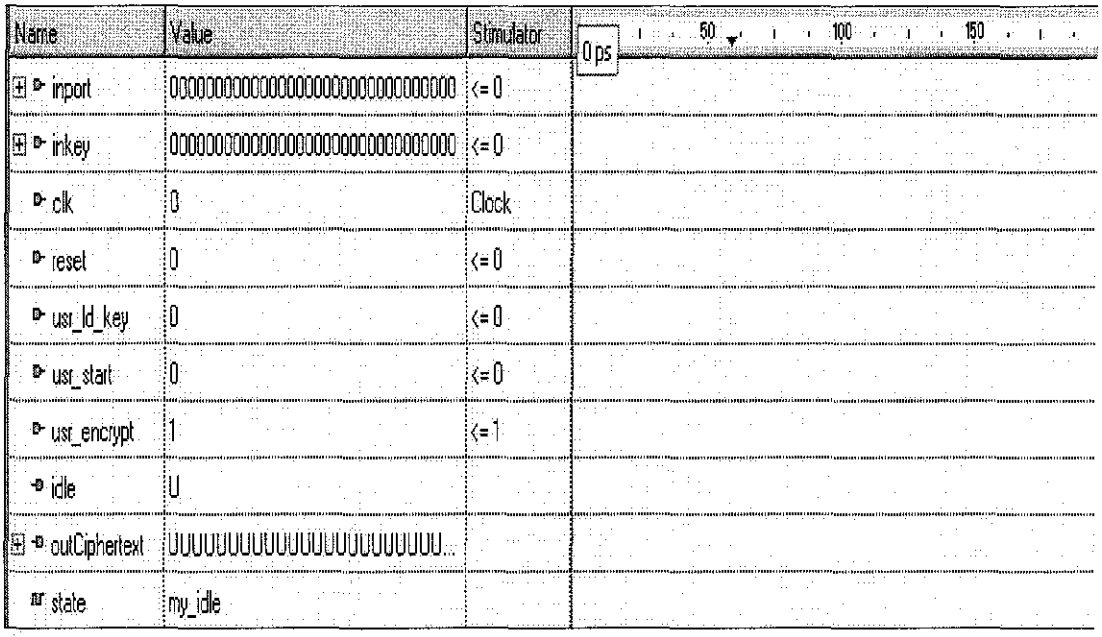


Figure 42: Idle – Text Vector 1

The values when idle.

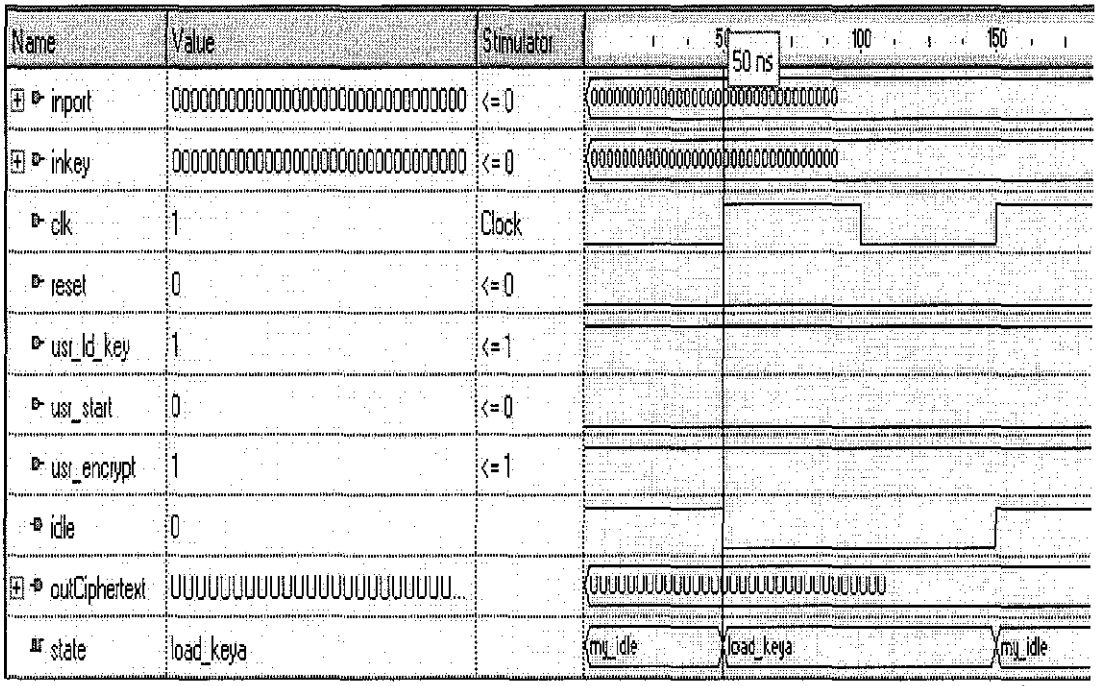


Figure 43: Load Key – Text Vector 1

The Key has been loaded. This is shown by latch_key = '1' after 50 ns. The state also proves that the state is Load Key. After the key has been loaded, the state becomes idle, waiting for the encryption process to start at 150 ns.

5.2.1.2 Start Encryption

Before we start this process, the following signals need to be set first.

- INPORT= 00000000000000000000000000000000 – (Plaintext)
- CLK = '1'
- USR_LD_KEY = '0'
- RESET = '0'
- USR_START ='1'
- USR_ENCRYPT ='1'

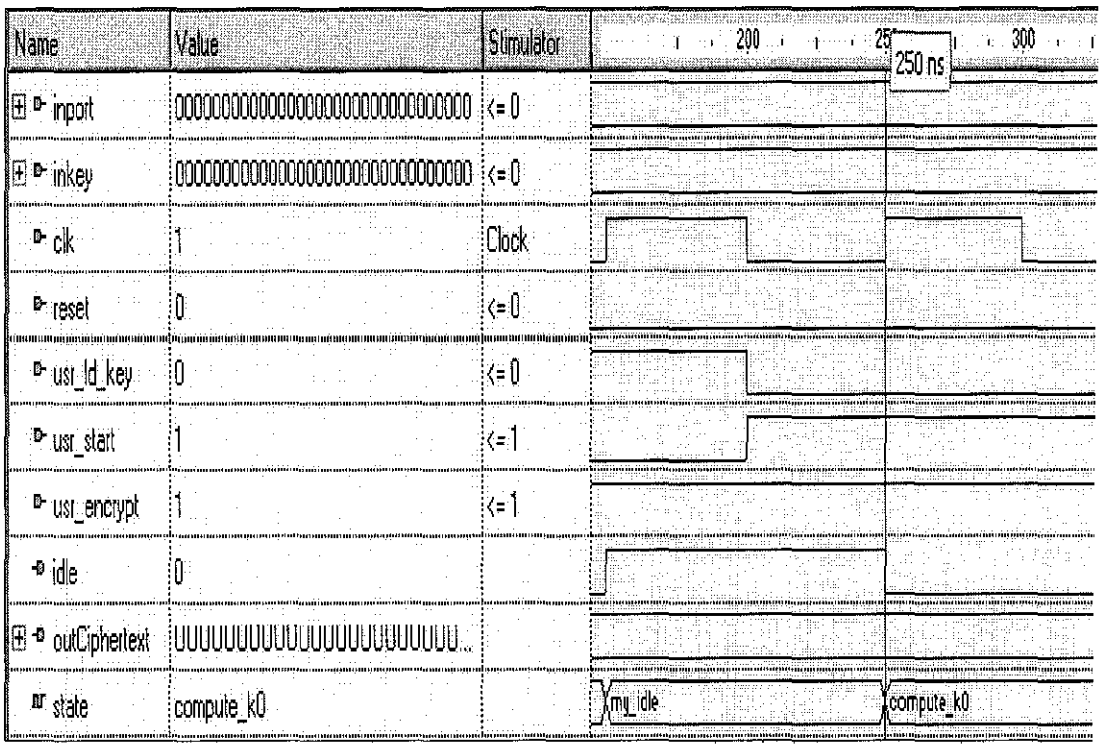


Figure 44: Start Encryption – Text Vector 1

We can see that the encryption starts at 250ns. This is proven by the change of the state to compute K0 indicating that K0 is being computed now.

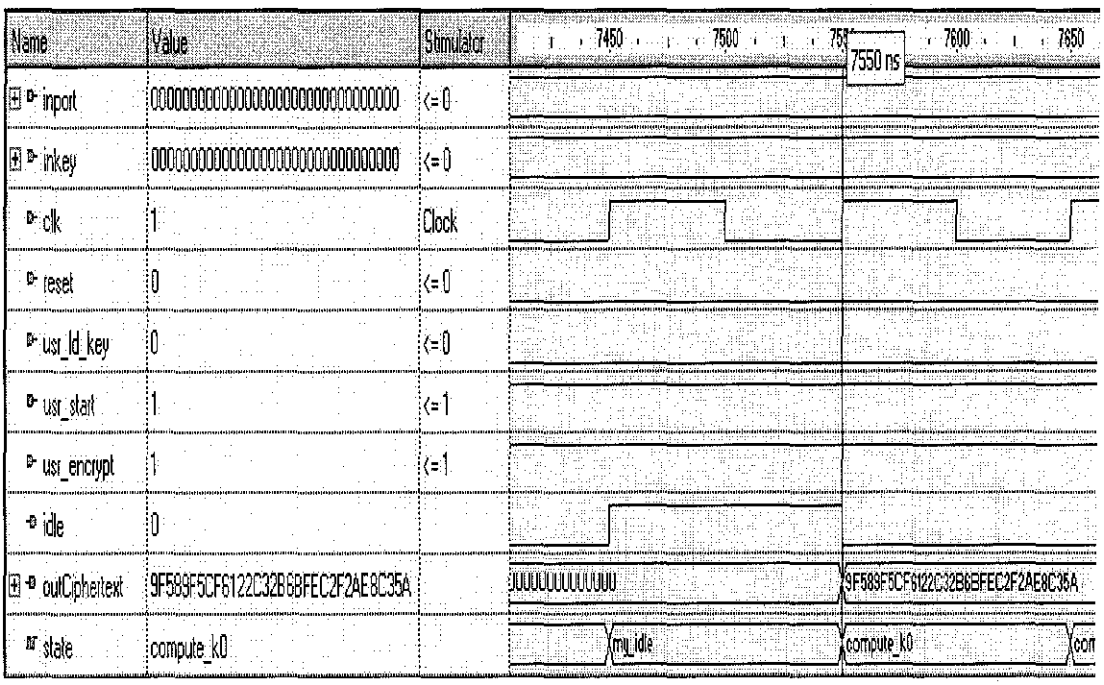


Figure 45: End of Encryption – Text Vector 1

As we can see above, the outCiphertext is obtained at 7550ns. The encrypted data obtained is **9F589F5CF6122C32B6BFEC2F2AE8C35A.** The output ciphertext obtained **matches with the expected value** given by the authors. One observation made is as follows:

Encryption duration = (7550-250) ns = **7300ns – 73 clock cycles**
One clock cycle = **100ns**
Expected encryption duration = **7200ns – 72 clock cycles**
Observation: **Delay of 100ns**

Justification

This can actually be justified. Actually, the whole encryption process was completed at 7450ns. This value actually corresponds to 72 clock cycles – expected value. A delay of 100ns is obtained due to locking the encrypted data into the output register. This locking takes one clock cycle and is done during idle state as seen above.

Therefore it is only fair to say that the total encryption time taken is 73 clock cycle.

5.2.2 Decryption

The decryption process is exactly opposite of the encryption process. Some changes need to be done. The changes are as follows:

KEY : 00000000000000000000000000000000 (maintains)
PLAINTEXT : 9F589F5CF6122C32B6BFEC2F2AE8C35A (encrypted data)
CIPHERTEXT: 00000000000000000000000000000000 (expected value)

The following are the main steps that need to be followed, before beginning the process.

- Load Key
- Start Decryption

5.2.2.1 Load Key

Before we start this process, the following signals need to be set first.

- INKEY= 00000000000000000000000000000000
- CLK = '1'
- USR_LD_KEY = '1'
- RESET = '0'
- USR_START ='0'
- USR_ENCRYPT ='0'

The clock speed that is used in the simulation is 10MHz.

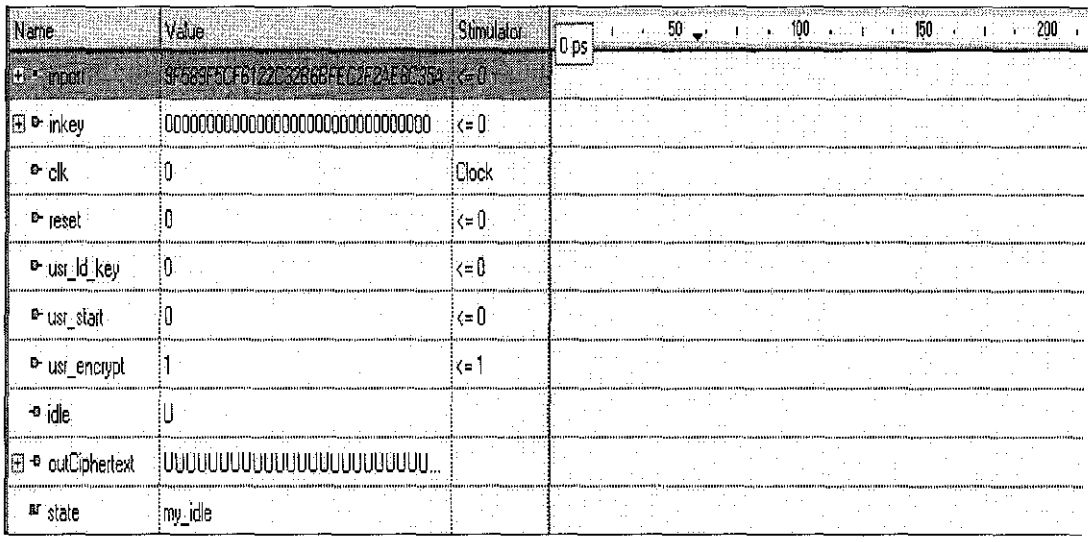


Figure 46: Initialization of Decryption – Text Vector 1

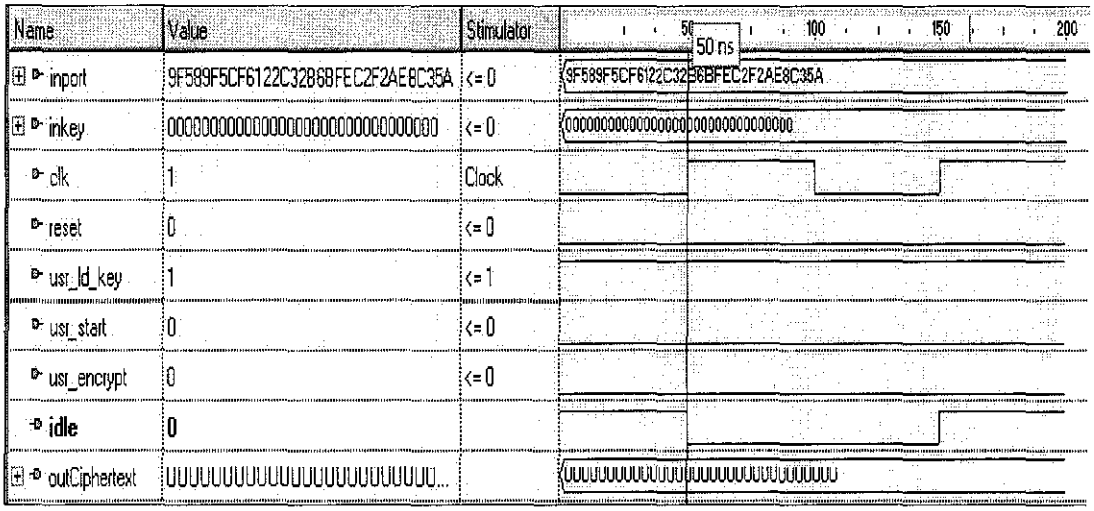


Figure 47: Decryption – Loading Key – Text Vector 1

The key has been loaded at 50ns.

5.2.2.2 Start of Decryption

Before we start this process, the following signals need to be set first.

- INPORT= 9F589F5CF6122C32B6BFEC2F2AE8C35A – (Plaintext)
- CLK = '1'
- USR_LD_KEY = '0'
- RESET = '0'
- USR_START ='1'
- USR_ENCRYPT ='0'

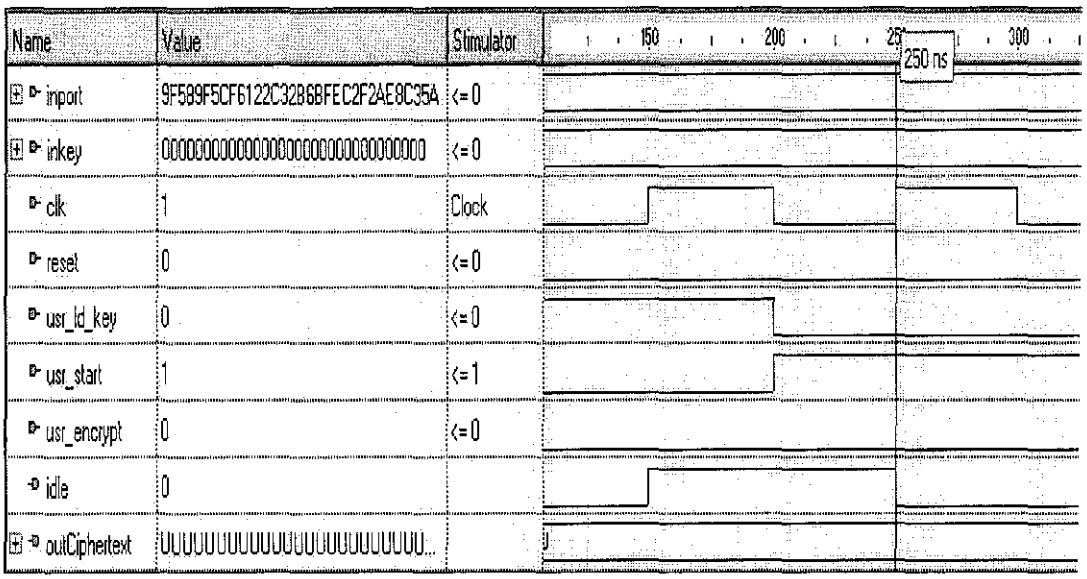


Figure 48: Start of Decryption – Text Vector

As can be seen above, decryption process started at 250ns.

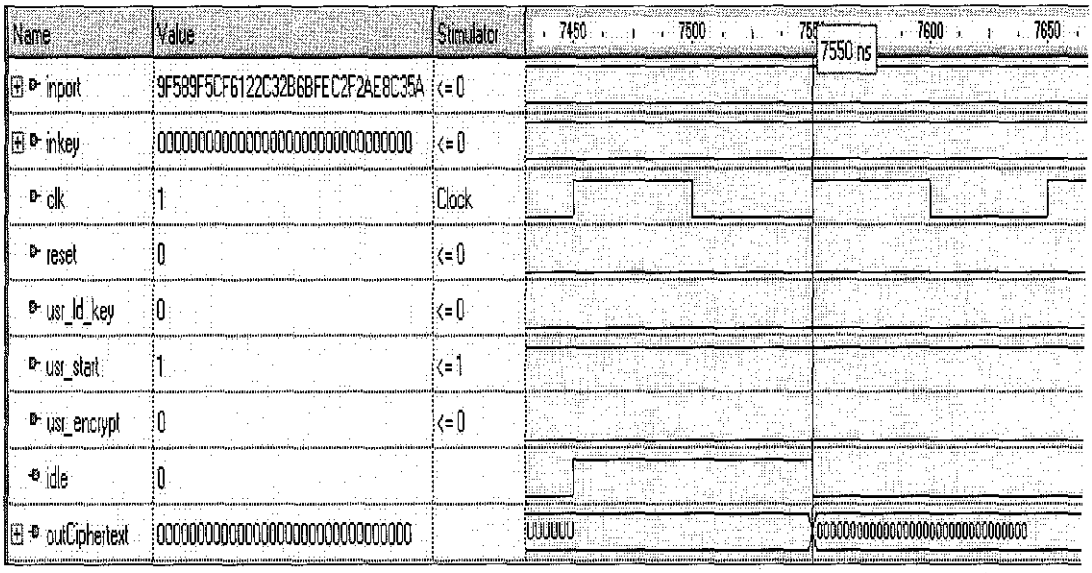


Figure 49: End of Decryption – Text Vector

From the above figure, it is true that the decrypted data obtained is **00000000000000000000000000000000. This is the expected value**. The decrypted data was obtained at 7550ns.

One observation made is as follows:

Decryption duration = (7550-250) ns = **7300ns – 73 clock cycles**

One clock cycle = **100ns**

Expected decryption duration = **7200ns – 72 clock cycles**

Observation: **Delay of 100ns**

Justification

This can actually be justified. Actually, the whole decryption process was completed at 7450ns. This value actually corresponds to 72 clock cycles – expected value. A delay of 100ns is obtained due to locking the decrypted data into the output register .This locking takes one clock cycle and is done during idle state as seen above.

Therefore it is only fair to say that the total decryption time taken is 73 clock cycle.

From the above simulation, we could make the following observation.

- The encryption cycle takes 73 clock cycles.
- The decryption cycle takes 73 clock cycles.
- Latency is 7300ns.

5.3 TEST VECTOR 2 – DESIGN 1

KEY : 137A24CA47CD12BE818DF4D2F4355960

PLAINTEXT : BCA724A54533C6987E14AA827952F921

CIPHERTEXT: 6B459286F3FFD28D49515B1581B08E42 (expected)

With this test vector, the simulation would be carried out for both encryption and decryption

5.3.1Encryption

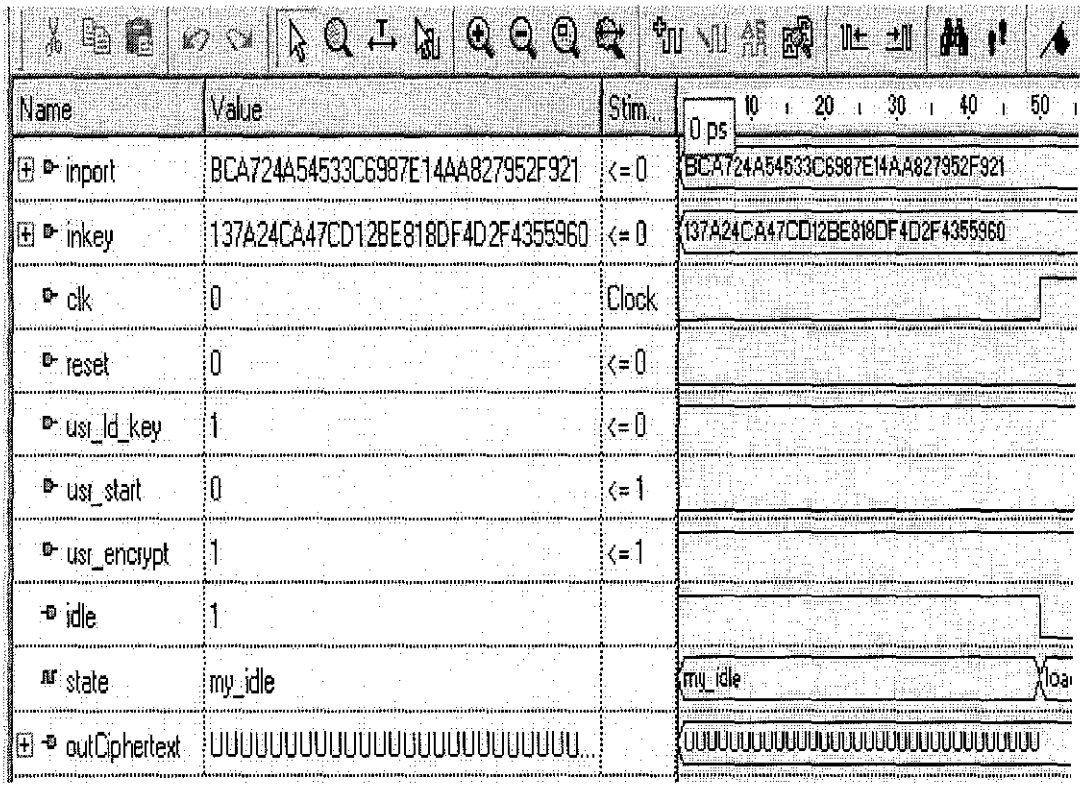


Figure 50: Initialization of Input Values – Test Vector 2

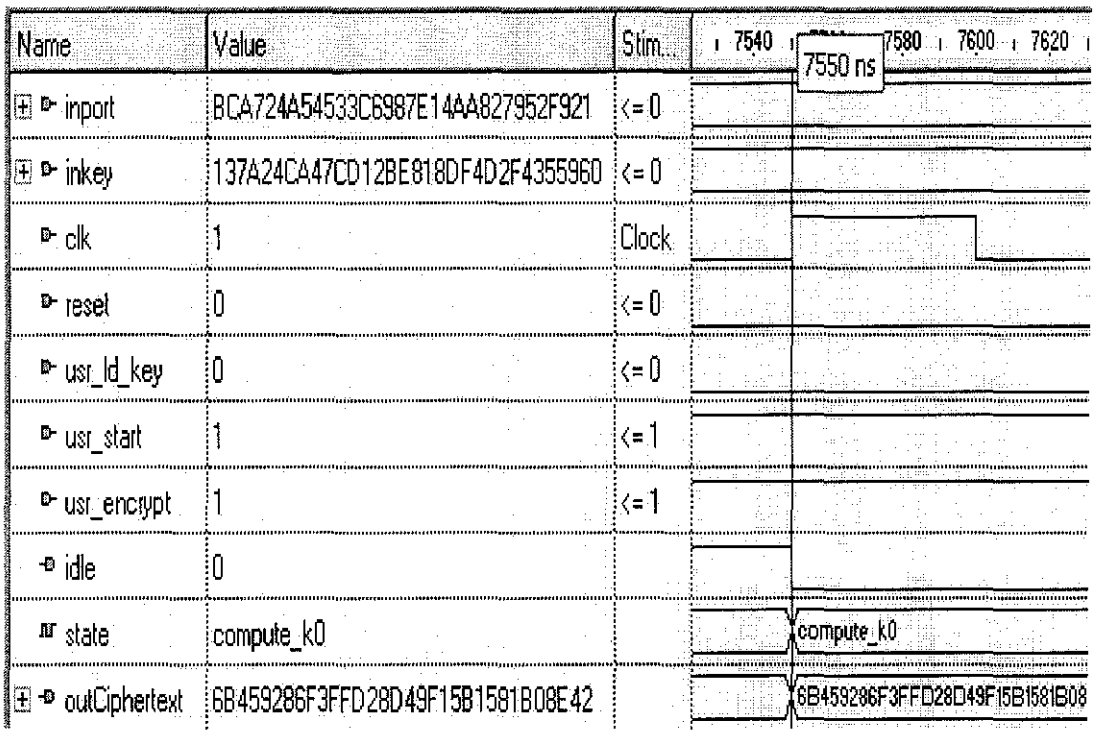


Figure 51: End of Encryption – Test Vector 2

As can be seen the output encrypted value tallies with the expected value. This proves that the system is behaving properly.

NOTE: Timing analysis was not considered for the second test vector – Only the output was considered.

5.3.2 Decryption

Decryption is exactly the opposite of the encryption. By using the same key and feeding the output value of the encryption process, the input of the encryption process would be obtained as the output of the decryption process.

KEY : 137A24CA47CD12BE818DF4D2F4355960

PLAINTEXT : 6B459286F3FFD28D49515B1581B08E42

CIPHERTEXT: BCA724A54533C6987E14AA827952F921 (expected)

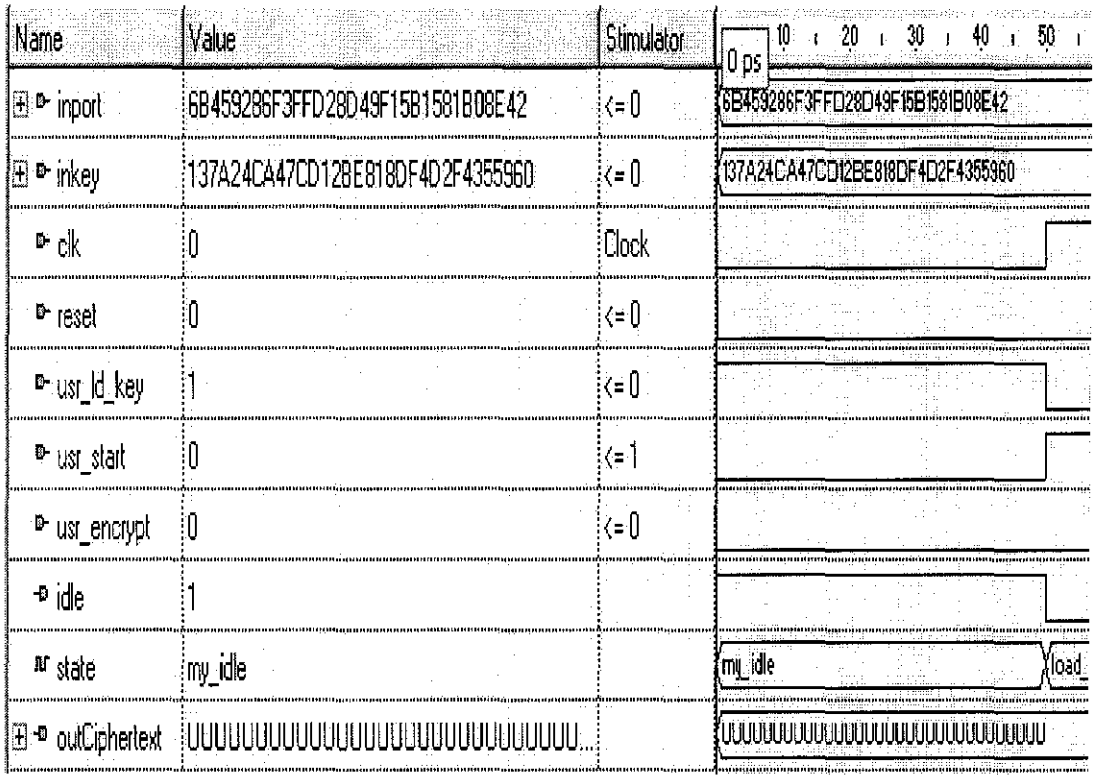


Figure 52: Initialization of Input Values – Test Vector 2

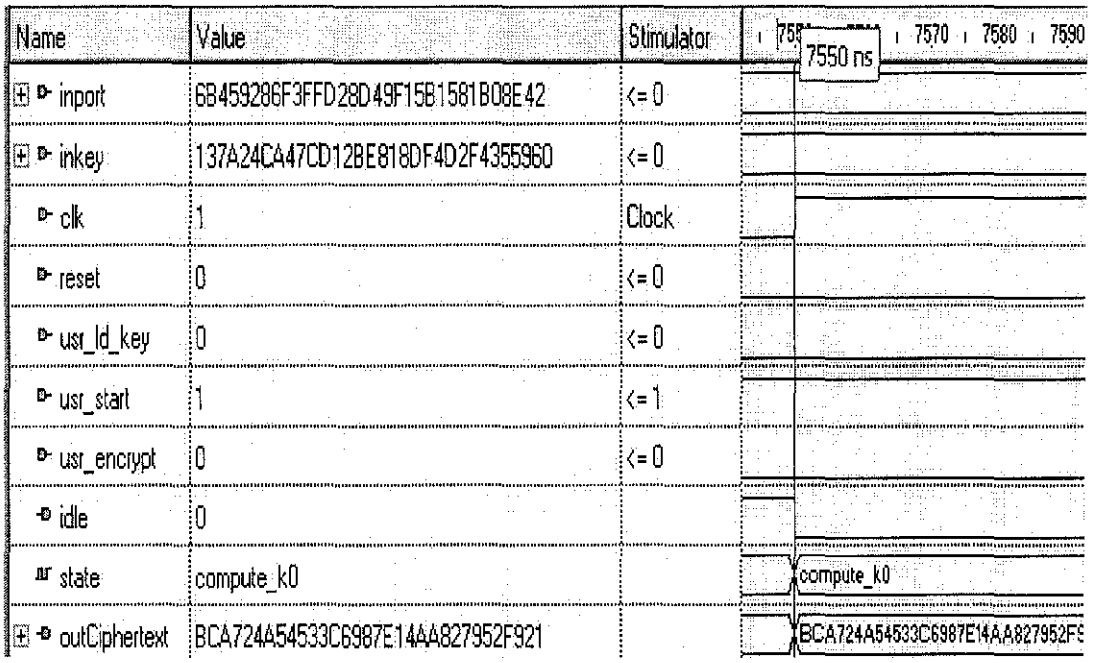


Figure 53: End of Decryption – Test Vector 2

As can be observed, the output values of the decryption process tallies with the expected output values. One interesting point to note here is with a given key, the decryption process is exactly the opposite of the encryption process.

5.4 TEST VECTOR 1 – DESIGN 2

As for Design 2, it is capable of performing both encryption and decryption too. The process is explained below.

The official test vector from the authors is as follows:-

KEY : 00000000000000000000000000000000
PLAINTEXT : 00000000000000000000000000000000
CIPHERTEXT: 9F589F5CF6122C32B6BFEC2F2AE8C35A (expected)

With this test vector, the simulation would be carried out for both encryption and decryption.

5.4.1Encryption

The following are the main steps that need to be followed, before beginning the process.

- Load Key
- Start Encryption

5.4.1.1 Load Key

Before we start this process, the following signals need to be set first.

- INKEY= 00000000000000000000000000000000
- CLK = '1'
- USR_LD_KEY = '1'
- RESET = '0'
- USR_START ='0'
- USR_ENCRYPT ='1'

The clock speed that is used in the simulation is 10MHz.

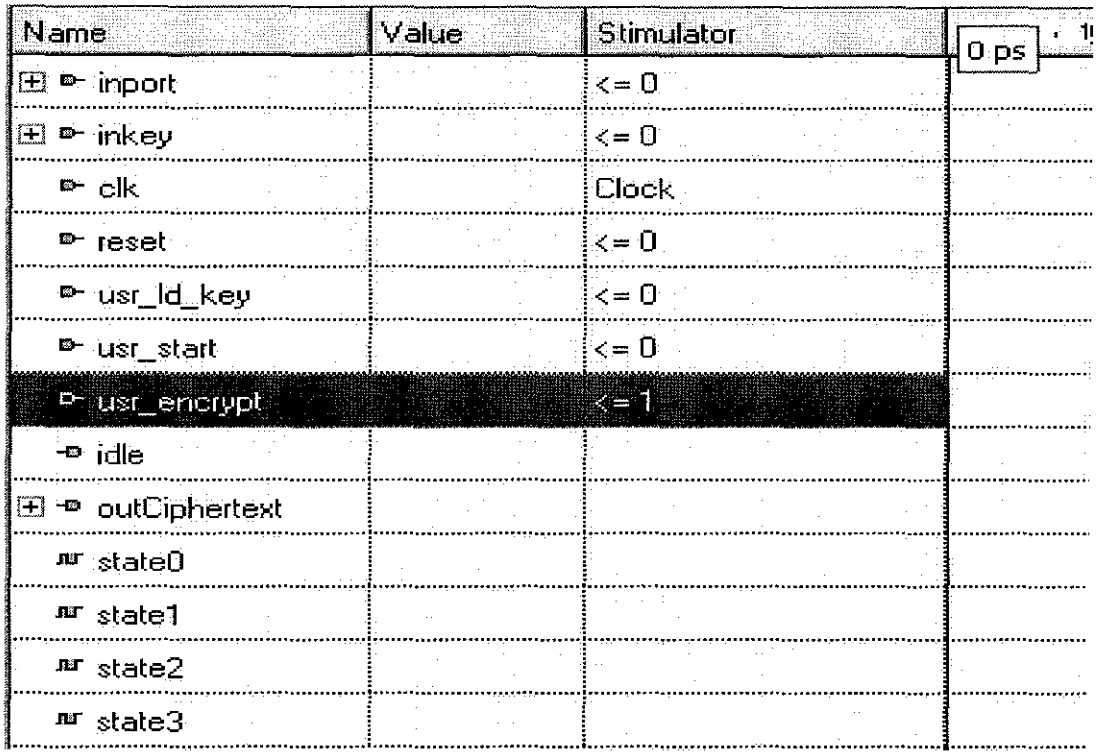


Figure 54: Idle – Text Vector 1

The values when idle.

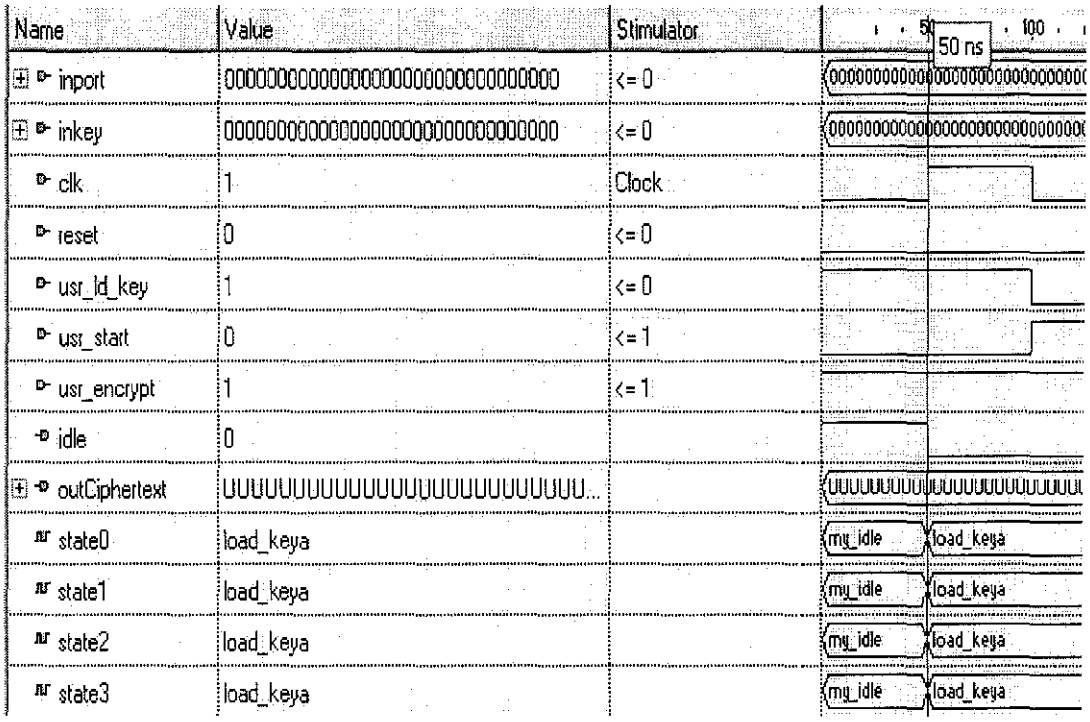


Figure 55: Load Key – Text Vector 1

The Key has been loaded. This is shown by the change of the 4 different states to Load_KeyA at 50ns indicating that the key has been loaded.

5.4.1.2 Start Encryption

Before we start this process, the following signals need to be set first.

- INPORT= 00000000000000000000000000000000 – (Plaintext)
- CLK = '1'
- USR_LD_KEY = '0'
- RESET = '0'
- USR_START ='1'
- USR_ENCRYPT ='1'

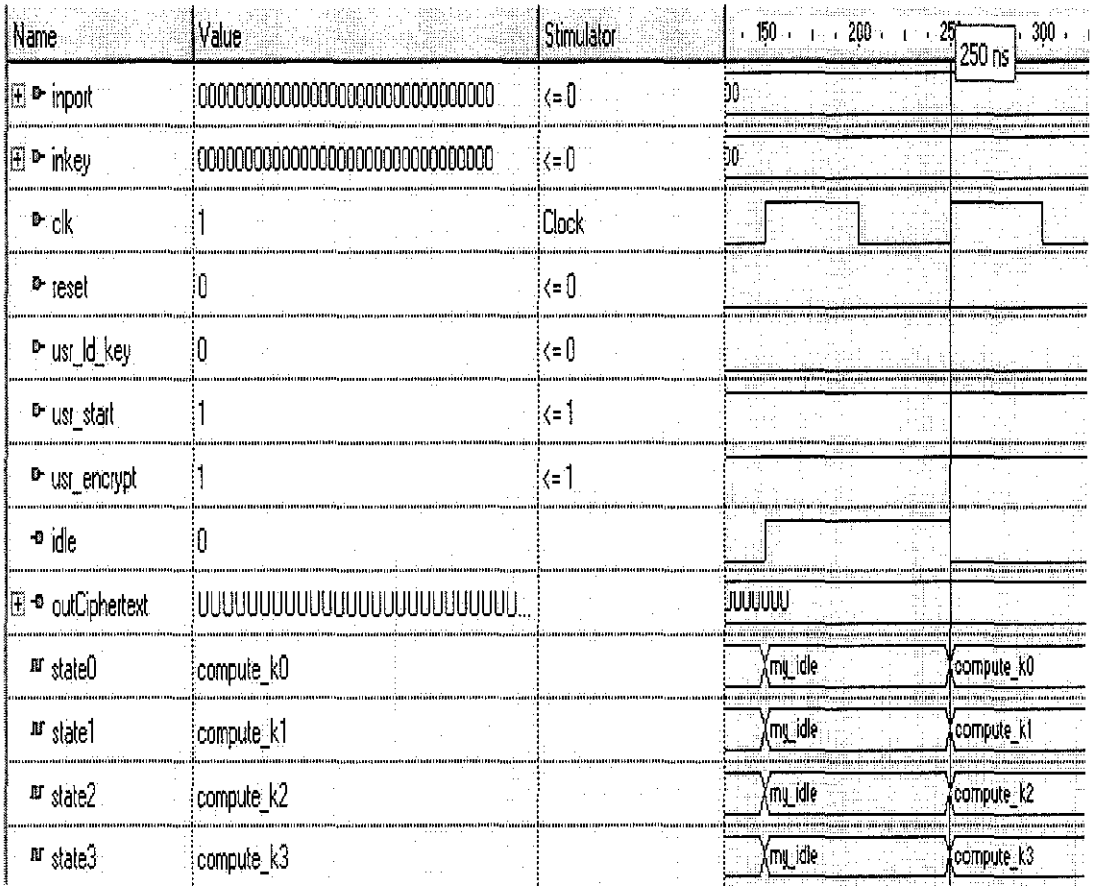


Figure 56: Start of Encryption – Text Vector 1

We can see that the encryption starts at 250ns. This is proven by the change of the 4 states to compute K0, compute K1, compute K2, and compute K3 indicating that K0, K1, K2 and K3 are being computed now.

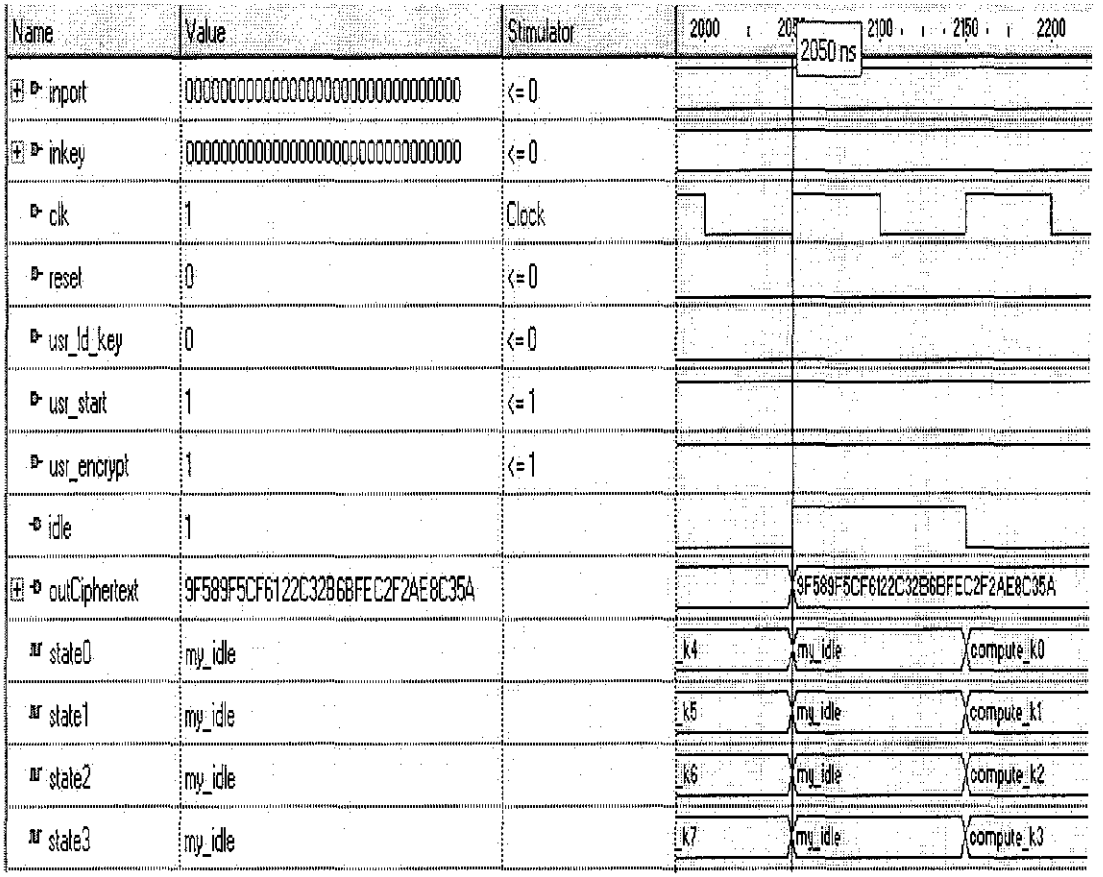


Figure 57: End of Encryption – Text Vector 1

As we can see above, the outCiphertext is obtained at 2050ns. The encrypted data obtained is **9F589F5CF6122C32B6BFEC2F2AE8C35A.** The output ciphertext obtained **matches with the expected value** given by the authors. One observation made is as follows:

Encryption duration = (2050-250) ns = **1800ns – 18 clock cycles**
One clock cycle = **100ns**
Expected encryption duration = **1800ns – 18 clock cycles**
Observation: **Matches As Expected**

It was observed that the output result obtained matches exactly with the result that was expected.

5.4.2 Decryption

The decryption process is exactly opposite of the encryption process. Some changes need to be done. The changes are as follows:

KEY : 00000000000000000000000000000000 (maintains)
PLAINTEXT : 9F589F5CF6122C32B6BFEC2F2AE8C35A (encrypted data)
CIPHERTEXT: 00000000000000000000000000000000 (expected value)

The following are the main steps that need to be followed, before beginning the process.

- Load Key
- Start Decryption

5.4.2.1 Load Key

Before we start this process, the following signals need to be set first.

- INKEY= 00000000000000000000000000000000
- CLK = '1'
- USR_LD_KEY = '1'
- RESET = '0'
- USR_START ='0'
- USR_ENCRYPT ='0'

The clock speed that is used in the simulation is 10MHz.

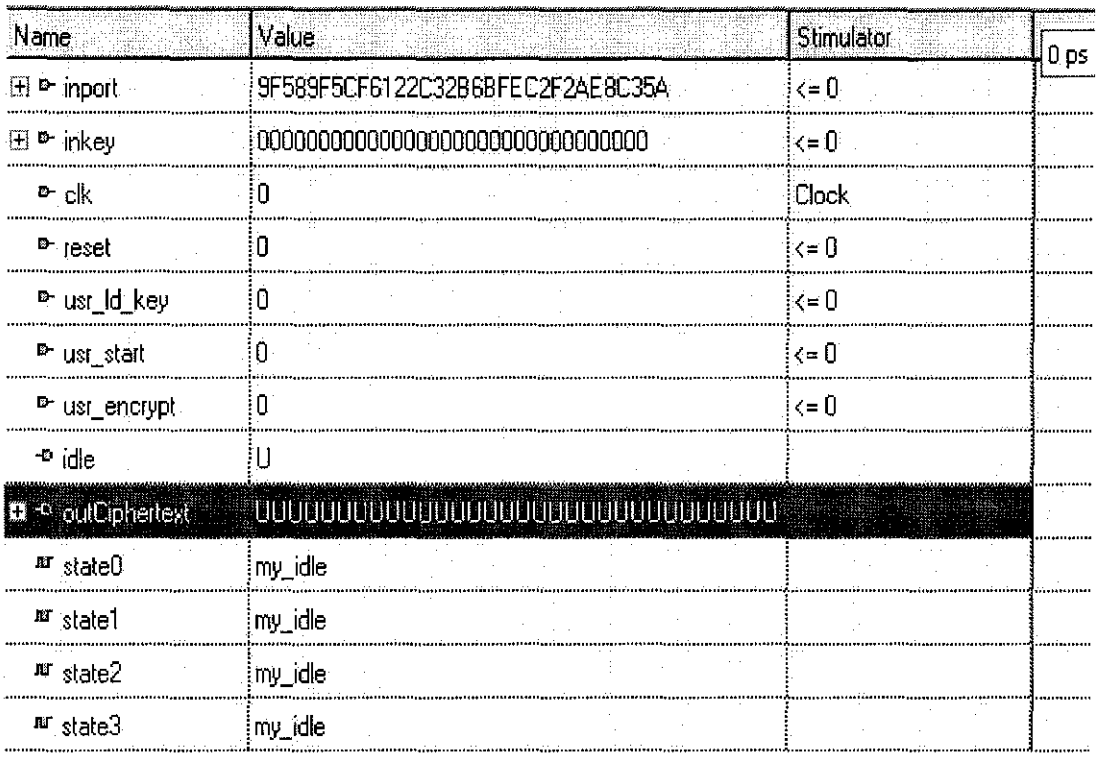


Figure 58: Initialization of Decryption – Text Vector 1

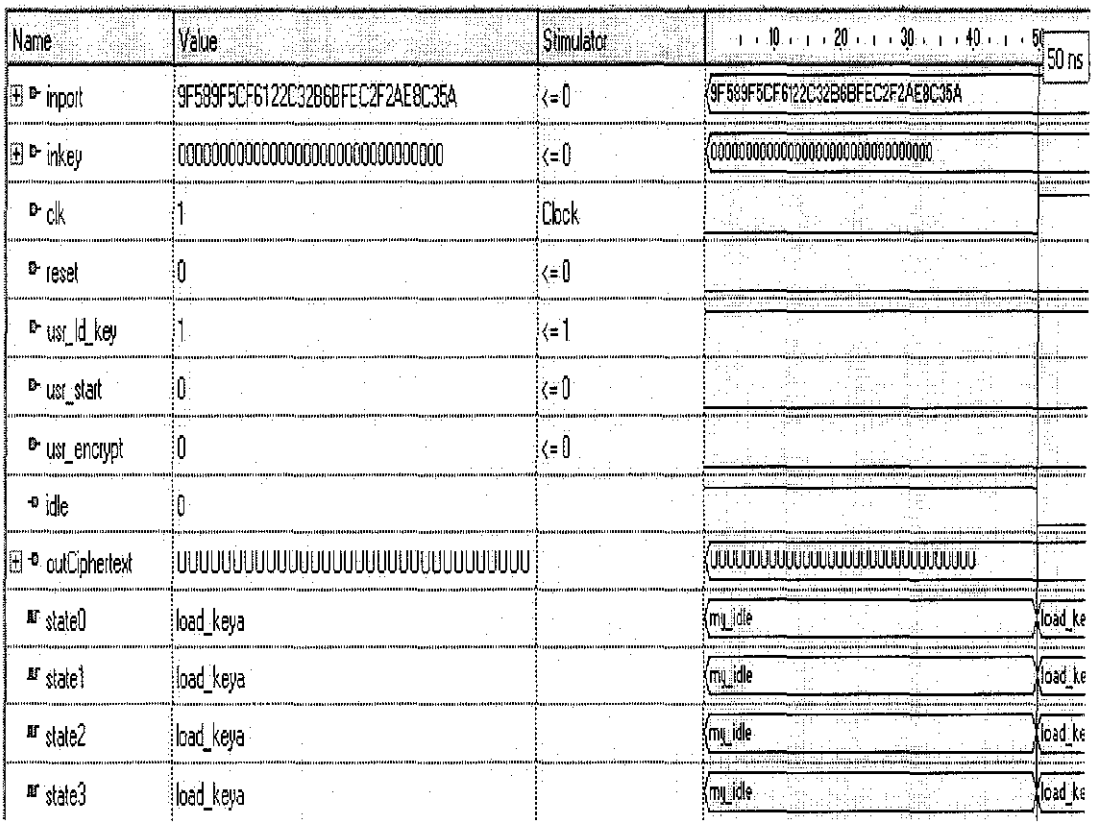


Figure 59: Decryption – Loading Key – Text Vector 1

The key has been loaded at 50ns.

5.4.2.2 Start of Decryption

Before we start this process, the following signals need to be set first.

- INPORT= 9F589F5CF6122C32B6BFEC2F2AE8C35A – (Plaintext)
- CLK = '1'
- USR_LD_KEY = '0'
- RESET = '0'
- USR_START ='1'
- USR_ENCRYPT ='0'

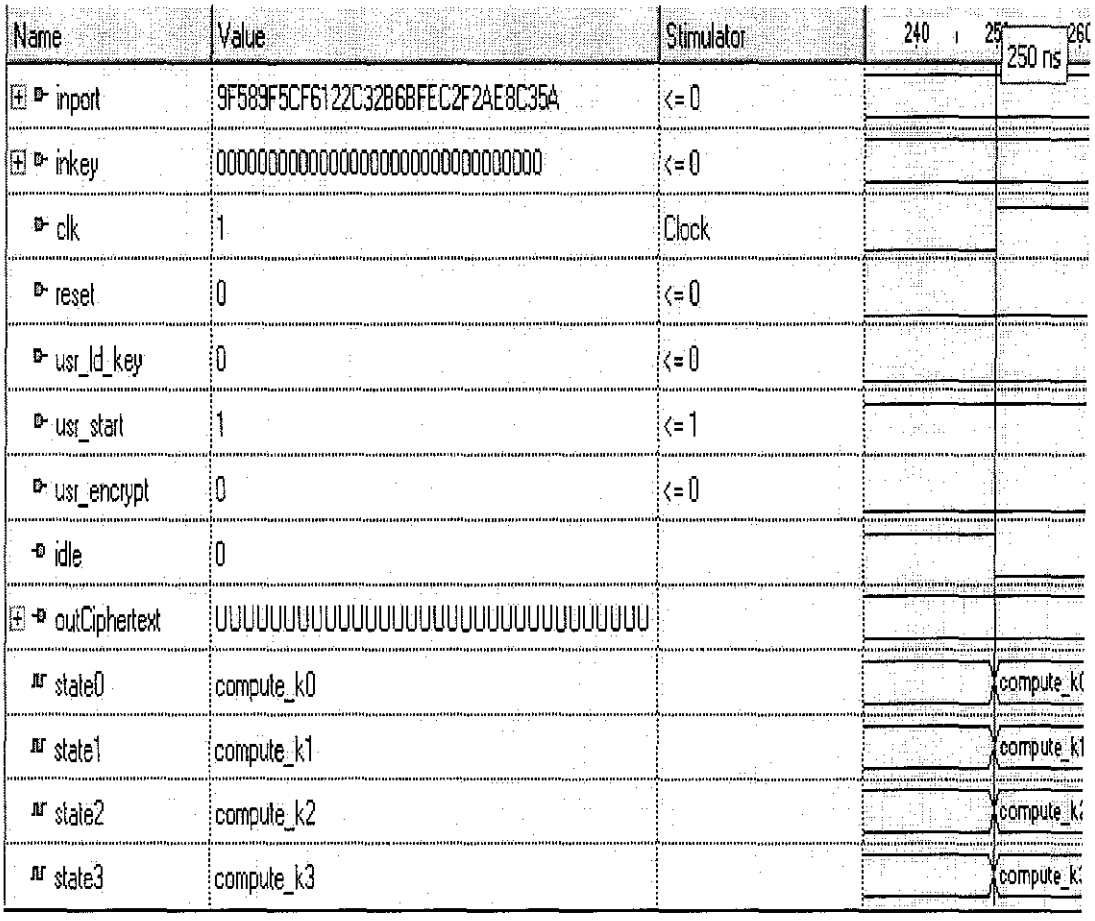


Figure 60: Start of Decryption – Text Vector

As can be seen above, decryption process started at 250ns.

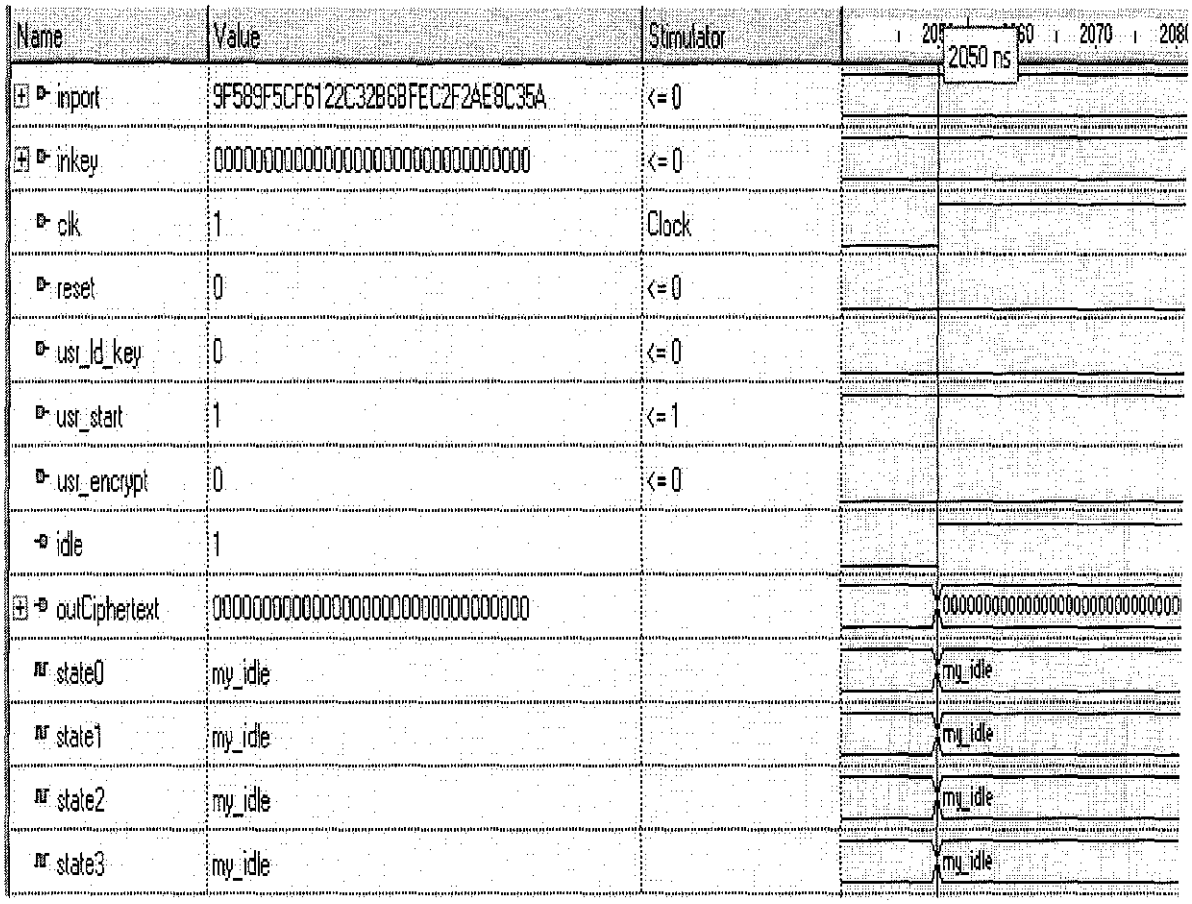


Figure 61: End of Decryption – Text Vector

From the above figure, it is true that the decrypted data obtained is **00000000000000000000000000000000. This is the expected value**. The decrypted data was obtained at 2050ns.

One observation made is as follows:

Decryption duration = (2050-250) ns = **1800ns – 18 clock cycles**

One clock cycle = **100ns**

Expected decryption duration = **1800ns – 18 clock cycles**

Observation: **Matches As Expected**

It was observed that the output result obtained matches exactly with the result that was expected.

From the above simulation, we could make the following observations.

- The encryption cycle takes 18 clock cycles.
- The decryption cycle takes 18 clock cycles.
- Latency is 1800ns.

5.5 TEST VECTOR 2 – DESIGN 2

KEY : 137A24CA47CD12BE818DF4D2F4355960

PLAINTEXT : BCA724A54533C6987E14AA827952F921

CIPHERTEXT: 6B459286F3FFD28D49515B1581B08E42 (expected)

With this test vector, the simulation would be carried out for both encryption and decryption

5.5.1Encryption

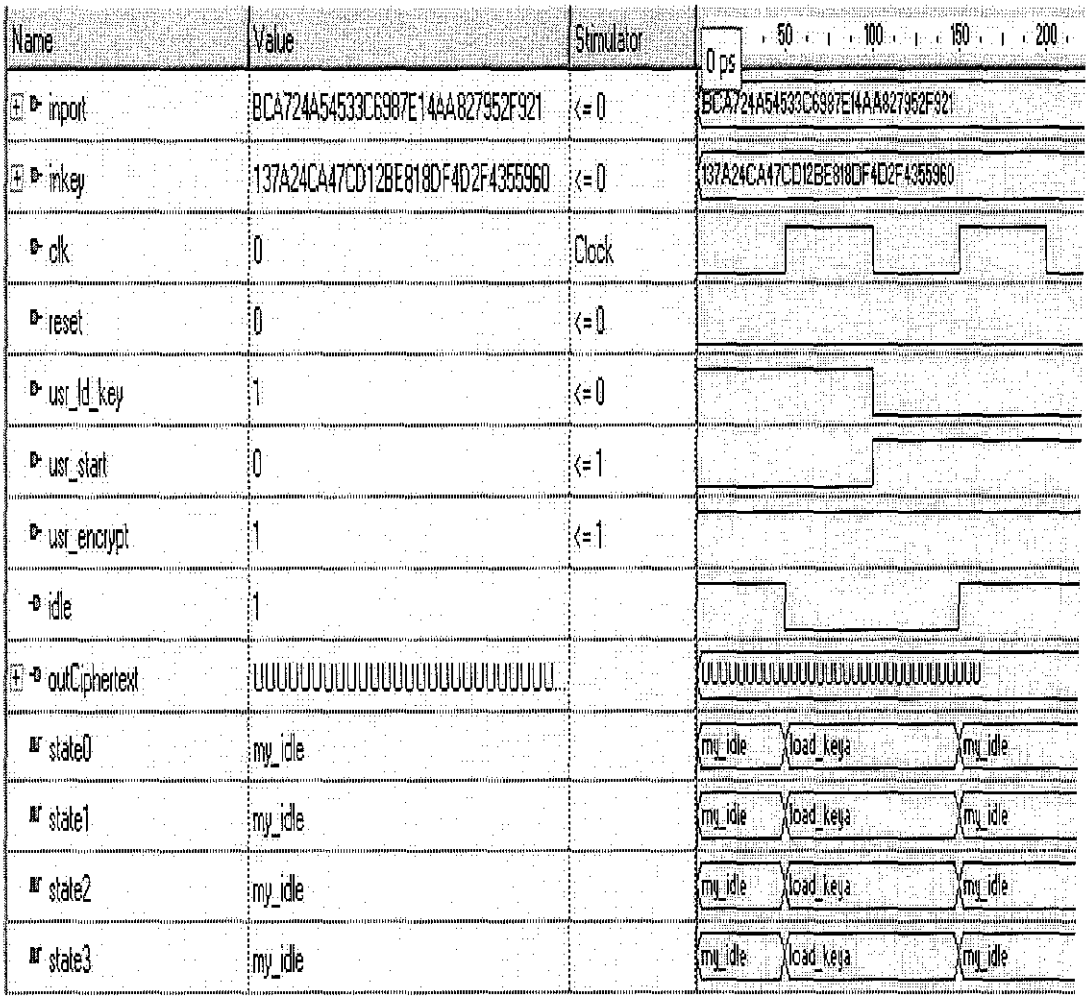


Figure 62: Initialization of Input Values – Test Vector 2

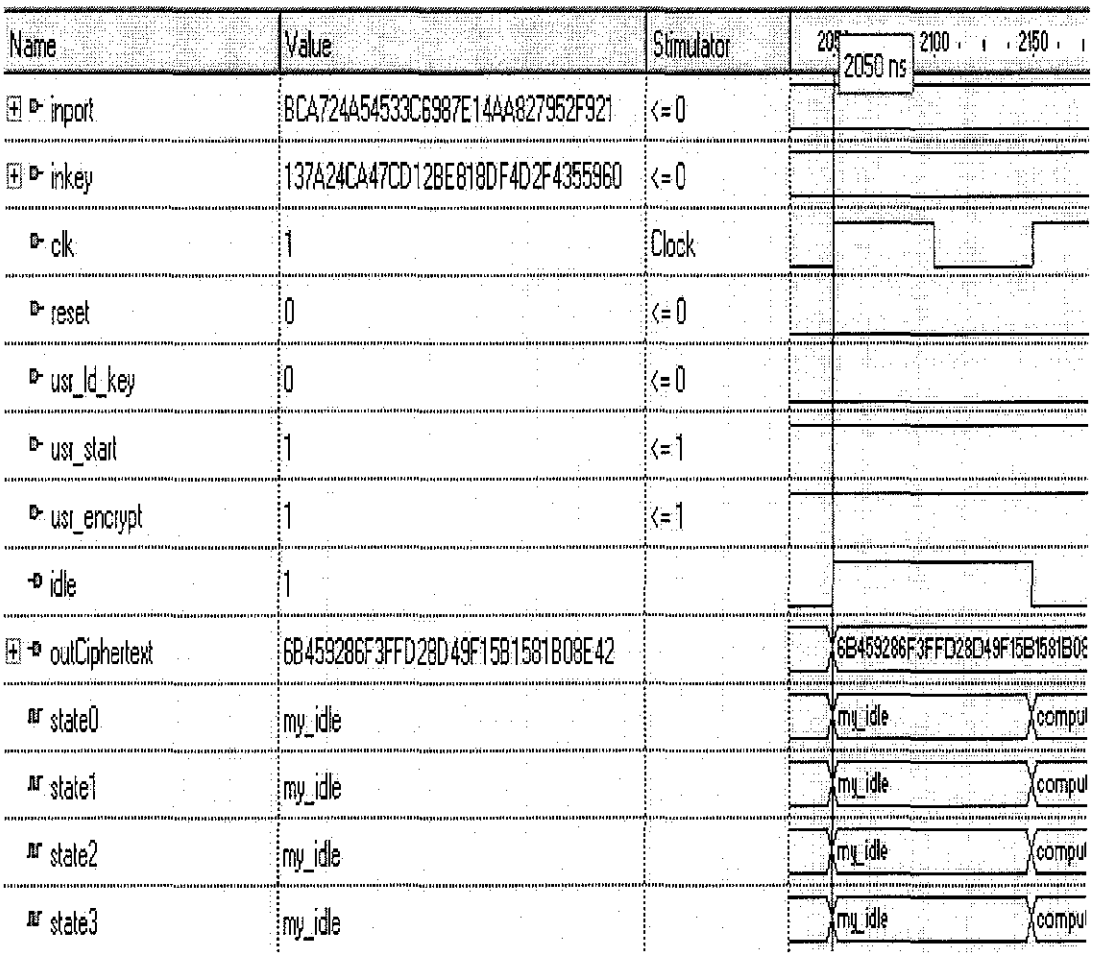


Figure 63: End of Encryption – Test Vector 2

As can be seen the output encrypted value tallies with the expected value. This proves that the system is behaving properly.

NOTE: Timing analysis was not considered for the second test vector – Only the output was considered.

5.5.2 Decryption

Decryption is exactly the opposite of the encryption. By using the same key and feeding the output value of the encryption process, the input of the encryption process would be obtained as the output of the decryption process.

KEY : 137A24CA47CD12BE818DF4D2F4355960

PLAINTEXT : 6B459286F3FFD28D49515B1581B08E42

CIPHERTEXT: BCA724A54533C6987E14AA827952F921 (expected)

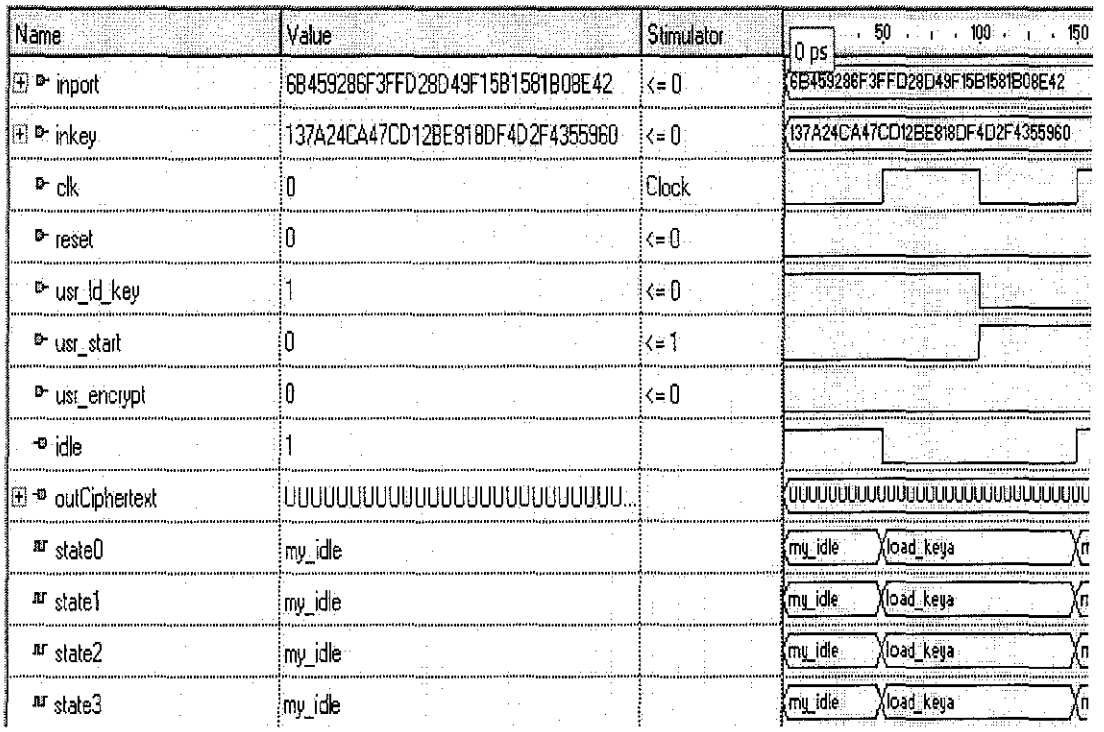


Figure 64: Initialization of Input Values – Test Vector 2

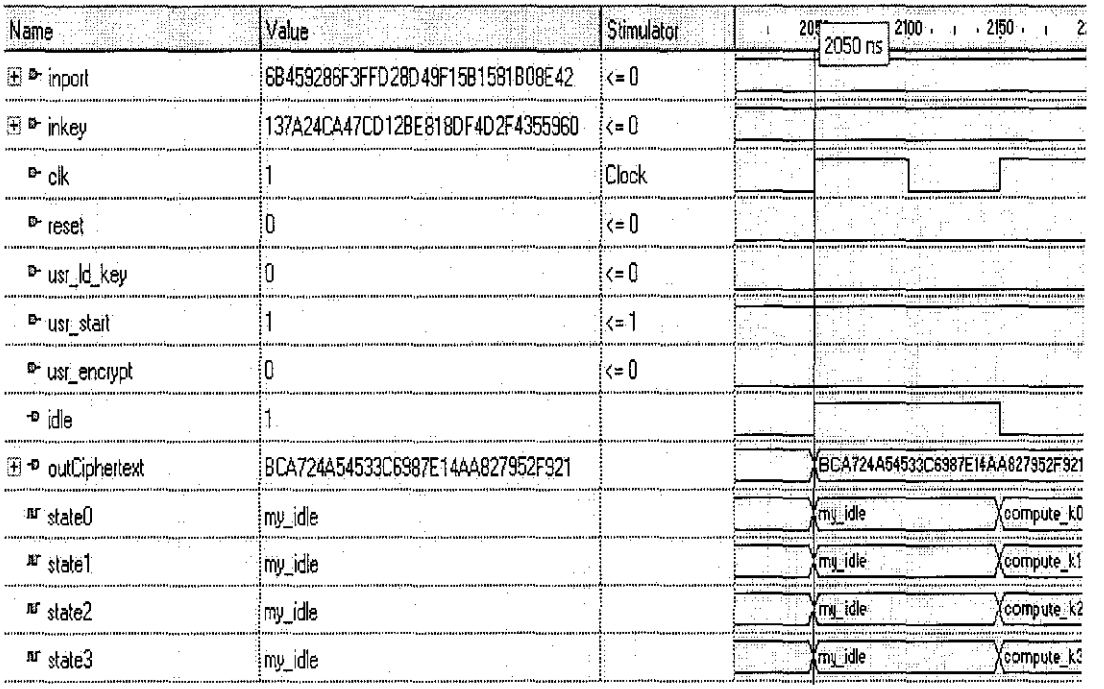


Figure 65: End of Decryption – Test Vector 2

As can be observed, the output values of the decryption process tallies with the expected output values. One interesting point to note here is with a given key, the decryption process is exactly the opposite of the encryption process.

NOTE: Timing analysis was not considered for the second test vector – Only the output was considered.

In general we can conclude both designs work perfectly without any flaw. Furthermore, due to heavy pipelining techniques adapted, the F_Functions of both Design 1 and Design 2 were never idle. This shows that the both designs are extremely optimized indeed. **For encryption and decryption process, the output that was obtained matched with the test vectors given by the authors.**

5.6 DESIGN IMPLEMENTATION

Once the simulation is completed, the design is ready to be implemented. The implementation is process is basically divided into 3 parts namely: -

- **Translate**

  This is the process of translating of reading NGO files, reading component libraries for design expansion, annotating constraints to design files, checking timing specifications and constraints, and checking expanded designs.

- **Map**

  It is the process of translating a design to available resources. It states what available resources are taken up by a given design.

- **Place and Route**

  This is the process of placing and routing a given design onto the FPGA.

A good design process during this process is to specify the pins of the FPGA for a given function. Eg. Pin 77 for CLOCK instead of letting the tools to decide. This is very useful in the latter stages when the device is used for application purposes or testing purposes. The **Spartan 2- XC2S200-5PQ208** FPGA chip found in UTP Lab was assembled by Burch Electronics. This company attached the FPGA to a custom made motherboard sold by this company. The name of this product is.B3-Spartan 2+. Therefore, it is very important for me to go through the datasheet before assigning the pins for the input and output of my design. The pin assignments for my designs are as follows: -

Table 19: Pin Assignment to the Corresponding I/O

PIN	PIN NUMBER
Address [4]	57
Address[3]	58
Address[2]	59
Address[1]	60
Address[0]	61
Read	62
Write	63
Reset	67
Enable	68
Load_Key	69
Start	70
Encrypt	71
Idle	74
Clock	77
Append	80
Output[15]	81
Output[14]	82
Output[13]	83
Output[12]	84
Output[11]	86
Output[10]	87
Output[9]	88
Output[8]	89
Output[7]	90
Output[6]	94
Output[5]	95
Output[4]	96
Output[3]	97
Output[2]	98
Output[1]	99
Output[0]	100
Input[15]	206
Input[14]	205
Input[13]	204
Input[12]	203
Input[11]	202
Input[10]	201
Input[9]	200
Input[8]	199
Input[7]	195
Input[6]	194
Input[5]	193
Input[4]	192
Input[3]	191
Input[2]	189
Input[1]	188
Input[0]	187

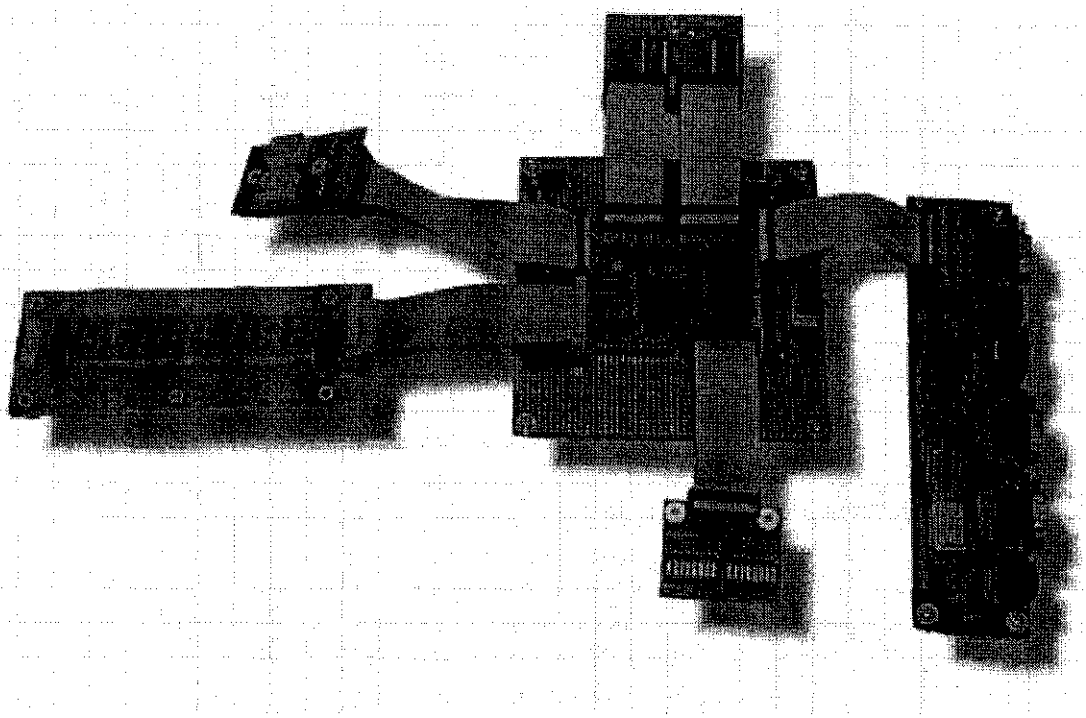


Figure 66: B3-SPARTAN2+ Board

5.7 GENERATE PROGRAMMING FILE & DOWNLOADING

The final step was to generate the programming file which is the ASCII Configuration file that has an extension of .rbt. Once that was done, the downloading process was performed by running the BEDLOAD utility. After setting the necessary parallel port settings, the design was downloaded.

CHAPTER 6: DISCUSSION

6.1 ORIGINAL DESIGN BY THE AUTHOR

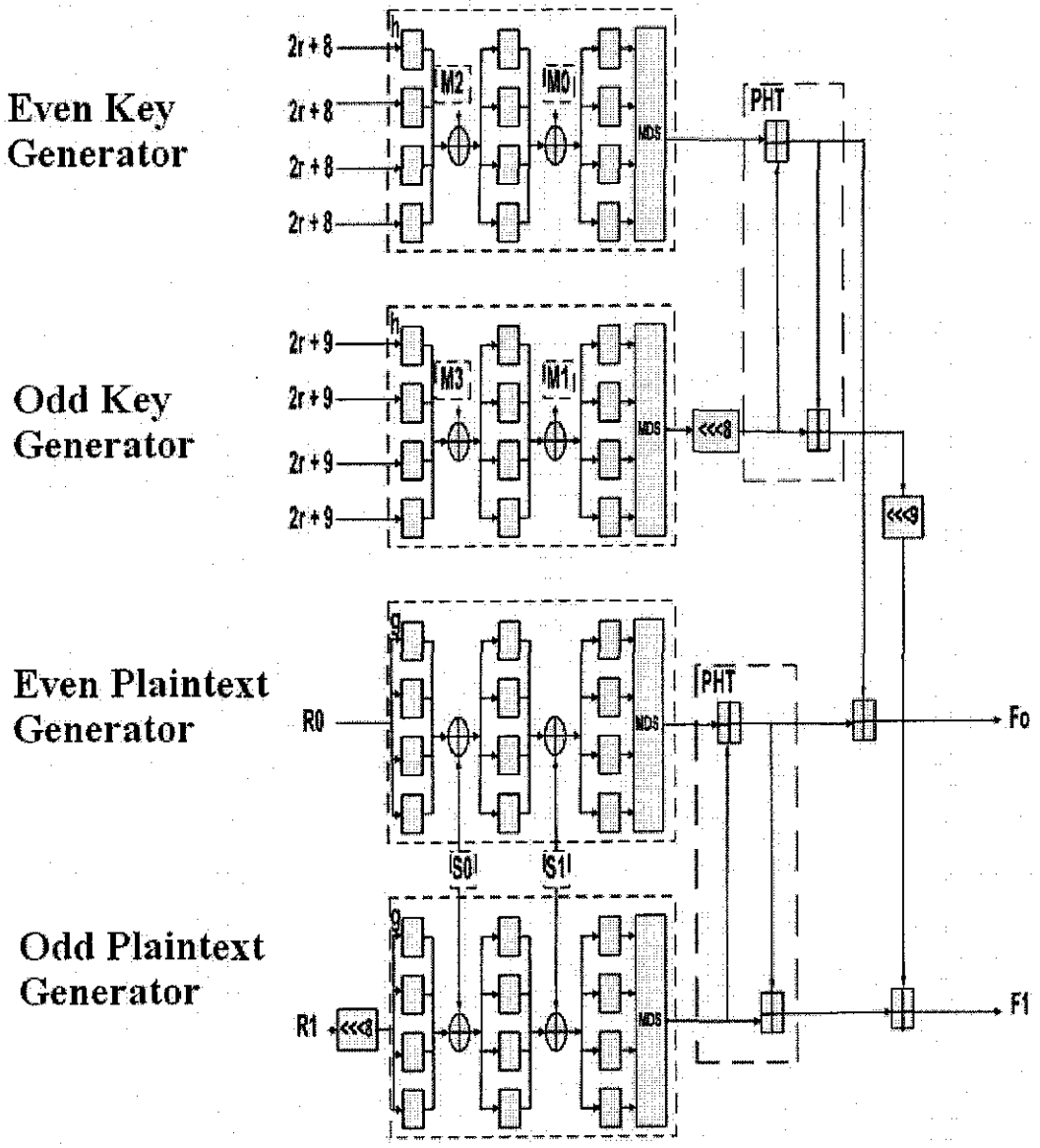


Figure 67: A View of a Complete Single Round F –Function of the Original Design as Proposed by the Author

***Assumption: All keys must be calculated on the fly and not to be pre-computed.**

The above design shows, that for a complete single round F – Function, we need to have the following: -

- Even Key Generator
- Odd Key Generator

- Even Plaintext Generator
- Odd Plaintext Generator

With the above design, when directly implemented on hardware could have the following sequence.

Table 20: Generated Sequence of Values Based on Design 1

Time Unit (at the end of this time unit)	Even Key Generator (Generated value)	Odd Key Generator (Generated value)	Even Plaintext Generator (Generated value)	Odd Plaintext Generator (Generated value)
1	K0	K1		
2	K2	K3		
3	K8	K9	P0 even	P0 odd
4	K10	K11	P1 even	P1 odd
5	K12	K13	P2 even	P2 odd
6	K14	K15	P3 even	P3 odd
7	K16	K17	P4 even	P4 odd
8	K18	K19	P5 even	P5 odd
9	K20	K21	P6 even	P6 odd
10	K22	K23	P7 even	P7 odd
11	K24	K25	P8 even	P8 odd
12	K26	K27	P9 even	P9 odd
13	K28	K29	P10 even	P10 odd
14	K30	K31	P11 even	P11 odd
15	K32	K33	P12 even	P12 odd
16	K34	K35	P13 even	P13 odd
17	K36	K37	P14 even	P14 odd
18	K38	K39	P15 even	P15 odd
19	K4	K5		
20	K6	K7		

Estimated latency for processing a block of data of 128 bits without wrapper = **20 clock cycles**

Estimated latency for processing a block of data of 128 bits with wrapper = **20 clock cycles**

NOTE: Wrapper built doesn't add any time delay because it uses pipelining technique.

6.2 DESIGN 1

Due to our design decisions of minimizing the hardware, the structure of the cipher had to be modified in order to be able to use the same h-function for all computation purposes. The following diagram shows the modified F-function that was designed in order to meet the requirements:

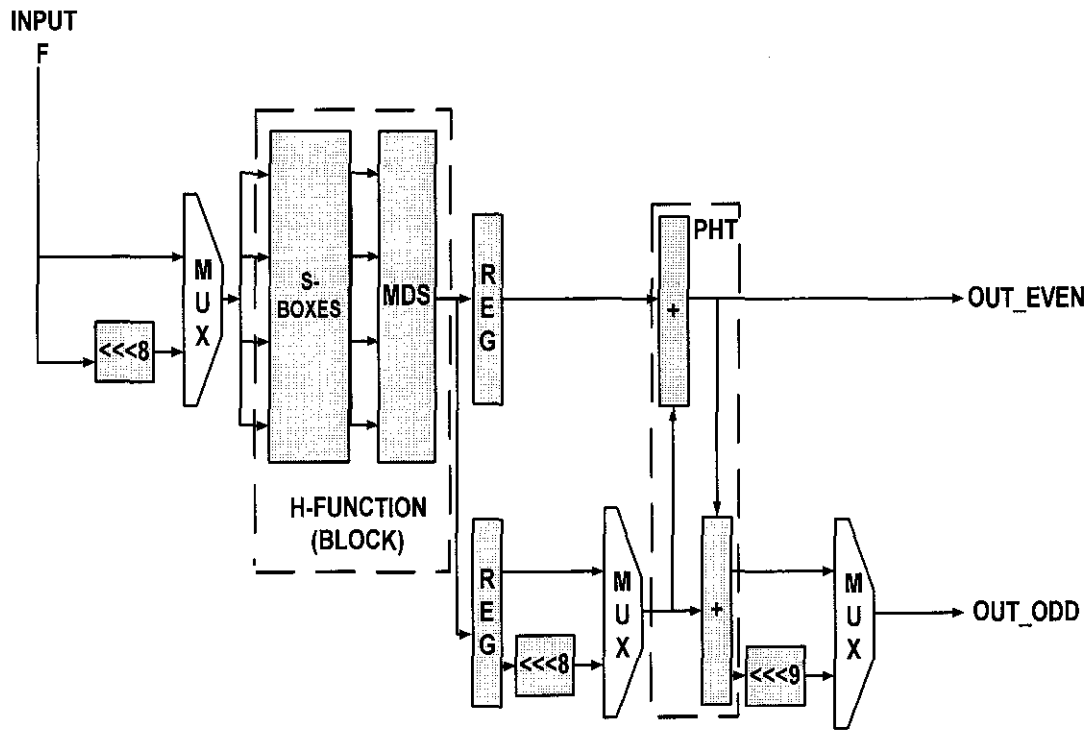


Figure 68: Generalized Modified F-Function

As can be observed, the input and a rotated version of the input are multiplexed to reproduce both inputs that can be presented at the input of an h-function. Moreover, in order to compute the result of the PHT, two registers had to be added to store the value of the first input or the former while the second was being computed. Thus, these two registers act as pipeline registers. The other two multiplexers added are used to choose between the path used for computing a key or the path used to encrypt or

decrypt data. This modified F- Function has the capability of performing the following: -

- Even Key Generation
- Odd Key Generation
- Even Plaintext Generation
- Odd Plaintext Generation

This is a revolutionary design because instead of using 4 separate parts, I have combined them into a single module which could perform all 4 functions. When directly implemented on hardware, the design could have the following sequence.

Table 21: Generated Sequence of Values Based on Design 1

Time Unit	Generated Value	Time Unit	Generated Value	Time Unit	Generated Value	Time Unit	Generated Value
1	K0	**19**	P3 even	**37**	K24	**55**	P12 even
2	K1	**20**	P3 odd	**38**	K25	**56**	P12 odd
3	K2	**21**	K16	**39**	P8 even	**57**	K34
4	K3	**22**	K17	**40**	P8 odd	**58**	K35
5	K8	**23**	P4 even	**41**	K26	**59**	P13 even
6	K9	**24**	P4 odd	**42**	K27	**60**	P13 odd
7	P0 even	**25**	K18	**43**	P9 even	**61**	K36
8	P0 odd	**26**	K19	**44**	P9 odd	**62**	K37
9	K10	**27**	P5 even	**45**	K28	**63**	P14 even
10	K11	**28**	P5 odd	**46**	K29	**64**	P14 odd
11	P1 even	**29**	K20	**47**	P10 even	**65**	K38
12	P1 odd	**30**	K21	**48**	P10 odd	**66**	K39
13	K12	**31**	P6 even	**49**	K30	**67**	P15 even
14	K13	**32**	P6 odd	**50**	K31	**68**	P15 odd
15	P2 even	**33**	K22	**51**	P11 even	**69**	K4
16	P2 odd	**34**	K23	**52**	P11 odd	**70**	K5
17	K14	**35**	P7 even	**53**	K32	**71**	K6
18	K15	**36**	P7 odd	**54**	K33	**72**	K7

Estimated latency for processing a block of data of = **72 clock cycles**

128 bits **without** wrapper

Estimated latency for processing a block of data of = **72 clock cycles**

128 bits **with** wrapper

NOTE: Wrapper built doesn't add any time delay because it uses pipelining technique.

When the above design was implemented using **Spartan 2- XC2S200-5PQ208**

Table 22: Output Performance of Design 1 on Spartan 2- XC2S200-5PQ208

Estimated Frequency	19.638MHz @ Period = 50.921ns
Latency	3.6663e-6s
Gate Count	26 318 gate counts
Throughput	34.9126Mbits/sec
Throughput/gate	1326.5667 Mbits/ s-gate

6.3 DESIGN 2

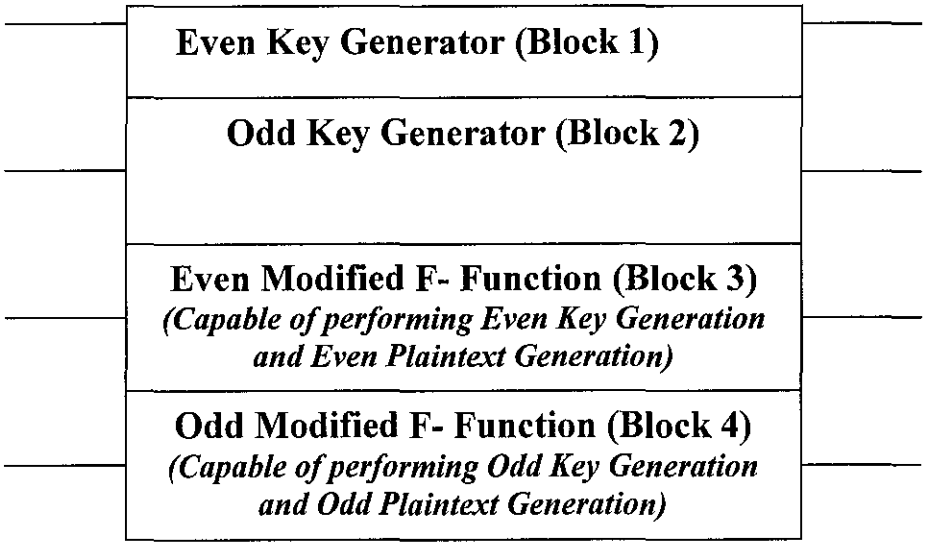


Figure 69: A View of a Complete Single Round F –Function of Design 2

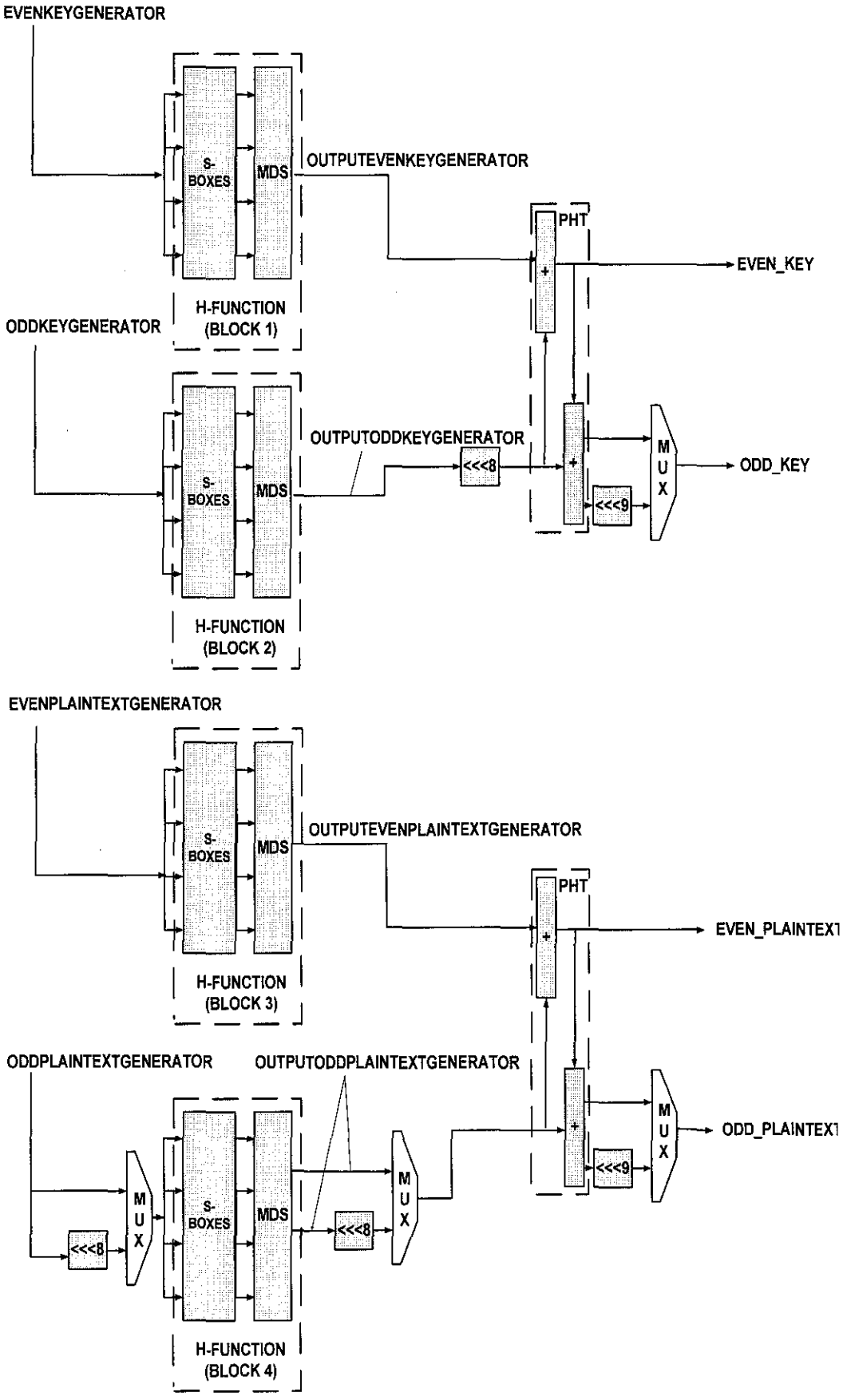


Figure 70: Generalized F-Function of Design 2

The above design shows, that for the Design 3, we need to have the following: -

- Even Key Generator
- Odd Key Generator
- Even Modified F- Function (capable of performing Even Key Generation and Even Plaintext Generation)
- Odd Modified F- Function (capable of performing Odd Key Generation and Odd Plaintext Generation)

This design looks quite similar to the original design which is the design proposed by the author. The difference is instead of using a dedicated Even and Odd Plaintext Generator that could only calculate plaintext values; I have decided to use a Modified F-Function which could calculate even and odd keys besides calculating even and odd plaintext. This design was borrowed from Design 1. After borrowing the design, further modification need to be made because the Modified F- Function in Design 1 could calculate all even and odd keys and even and odd plaintext. This would not be efficient and a waste of hardware resources for Design 2. I decided to split the function of Design 1 into smaller sub modules so that it could be incorporated into this design. As a result I obtained Even Modified F- Function and Odd Modified F-Function. The Even Modifed F-Function is capable of calculating even key and plaintext values and Odd Modified F-Function is capable of calculating odd key and plaintext values.

This is a far more efficient design because it incorporates the best of the Original Design (Author's Design) and Design 1. The major intentions are as follows:-

- A more efficient design.
- A very small latency.
- A very high throughput.
- A very high throughput/gate.
- A very minimal increase in hardware resources.

With the above design, when directly implemented on hardware has the following sequence.

Table 23: Generated Sequence of Values Based on Design 2

Time Unit (at the end of this time unit)	Even Key Generator	Odd Key Generator	Even Modified F-Function	Odd Modified F-Function
1	K0	K1	K2	K3
2	K8	K9	P0 even	P0 odd
3	K10	K11	P1 even	P1 odd
4	K12	K13	P2 even	P2 odd
5	K14	K15	P3 even	P3 odd
6	K16	K17	P4 even	P4 odd
7	K18	K19	P5 even	P5 odd
8	K20	K21	P6 even	P6 odd
9	K22	K23	P7 even	P7 odd
10	K24	K25	P8 even	P8 odd
11	K26	K27	P9 even	P9 odd
12	K28	K29	P10 even	P10 odd
13	K30	K31	P11 even	P11 odd
14	K32	K33	P12 even	P12 odd
15	K34	K35	P13 even	P13 odd
16	K36	K37	P14 even	P14 odd
17	K38	K39	P15 even	P15 odd
18	K4	K5	K6	K7

Estimated latency for processing a block of data of 128 bits **without** wrapper = **18 clock cycles**

Estimated latency for processing a block of data of 128 bits **with** wrapper = **18 clock cycles**

NOTE: Wrapper built doesn't add any time delay because it uses pipelining technique.

When the above design was implemented using **Spartan 2- XC2S200-5PQ208**

Table 24: Output Performance of Design 2 on Spartan 2- XC2S200-5PQ208

Estimated Frequency	8.0107MHz @ Period = 124.8420ns
Latency	2.2472e-6 s
Gate Count	32 616 gates
Throughput	56.9609Mbits/s
Throughput/gate	1746.4097Mbits/ s-gate

NOTE: Spartan 2 - XC2S200-5PQ208 is available in UTP Compute System Research Laboratory. Design has been successfully downloaded into the board.

6.4 PERFORMANCE DIFFERENCE BETWEEN DESIGN 1 AND DESIGN 2 ON SPARTAN 2- XC2S200-5PQ208

Design 1 would be chosen as the base value.

Table 25: Performance Difference of Design 1 and Design 2 on Spartan 2- XC2S200-5PQ208

	Design 1	**Design 2**	**% Difference**
Estimated Frequency	19.638MHz @ Period = 50.921ns	8.0107MHz @ Period = 124.842ns	Frequency = - 59.2 % Period = + 145.2%
Latency	3.6663e-6s	2.2472e-6 s	- 38.7%
Gate Count	26 318 gate counts	32 616 gates	+ 23.9%
Throughput	34.9126 Mbits/s	56.9609 Mbits/s	+ 63.2%
Throughput/gate	1326.5667 Mbits/ s-gate	1746.4097 Mbits/ s-gate	+ 31.6%

From the above design, we could see that Design 2 offers significance performance improvement. This is because optimizes all modules without leaving any modules from being idle. Among the significant improvement is the throughput which increased by 63.2% and the reduction in latency by 38.7%. Besides that the throughput/gate also increased significantly to1746.4097 Mbits/s-gate as compared to 1326.5667 Mbits/s-gate.

6.5 PERFORMANCE OF DESIGN 1 AND DESIGN 2 WHEN IMPLEMENTED ON SPARTAN 3 –XC3S400-5FG456

NOTE: Spartan3-XC3s400-5FG456 is a newer generation FPGA of the Spartan FPGA family. The designs are implemented on this FPGA for benchmarking purposes. This FPGA is not available in UTP.

The results obtained are as follows:

Table 26: Performance Difference of Design 1 and Design 2 on Spartan 3 –XC3S400-5FG456

	Design 1	Design 2	%Difference
Estimated Frequency	42.739MHz @ Period = 23.398ns	16.217MHz @ Period = 61.663ns	Frequency = - 60.64% Period = +163.5%
Latency	1.684656e-6 s	1.1099e-6 s	-34.1%
Gate Count	27 497 gates	33 044 gates	+20.17%
Throughput	75.9842 Mbits/s	115.3257 Mbits/s	+51.8%
Throughput/gate	2763.3644 Mbits/ s-gate	3490.0649 Mbits/ s-gate	26.3%

Overall, we could see that when Design 1 and Design 2 were implemented on Spartan 3–XC3S400-5FG456, tremendous performance improvements were achieved. The comparison between Spartan 2 and Spartan 3 performances would be discussed below.

6.6 PERFORMANCE COMPARISON OF DESIGN 1 WHEN IMPLEMENTED ON SPARTAN 2 - XC2S200-5PQ208 & SPARTAN 3 –XC3S400-5FG456

Implementation on Spartan 2 would be the base.

Table 27: Performance Difference of Design 1 on Spartan 2 - XC2S200-5PQ208 & Spartan 3 – XC3S400-5FG456

	Spartan 2	Spartan 3	% Difference
Estimated Frequency	19.638MHz @ Period = 50.921ns	42.739MHz @ Period = 23.398ns	Frequency = + 117.6% Period = -54.1%
Latency	3.6663e-6s	1.684656e-6 s	- 54.1%
Gate Count	26 318 gate counts	27 497 gates	+4.5 %
Throughput	34.9126 Mbits/sec	75.9842 Mbits/s	+ 117.6%
Throughput/gate	1326.5667 Mbits/ s-gate	2763.3644 Mbits/ s-gate	+ 108.3%

Significant performance improvements were achieved in frequency, latency, throughput and throughput/gate. A brief glance shows performance improvement of 2 times. Frequency increased by 117.6% to 42.739MHz. Besides that, throughput also increased more than 2 times to 75.9842Mbits/s from 34.9126 Mbits/sec.

6.7 Performance Comparison of Design 2 when Implemented on Spartan 2 - XC2S200-5PQ208 & Spartan 3 –XC3S400-5FG456

Implementation on Spartan 2 would be the base.

Table 28: Performance Difference of Design 2 on Spartan 2 - XC2S200-5PQ208 & Spartan 3 – XC3S400-5FG456

	Spartan 2	Spartan 3	% Difference
Estimated Frequency	8.0107MHz @ Period = 124.842ns	16.217MHz @ Period = 61.663ns	Frequency = + 117.1% Period = -50.6%
Latency	2.2472e-6 s	1.1099e-6 s	- 50.6%
Gate Count	32 616 gates	33 044 gates	+1.3 %
Throughput	56.9609 Mbits/s	115.3257 Mbits/s	+ 102.5%
Throughput/gate	1746.4097 Mbits/ s-gate	3490.0649 Mbits/ s-gate	+ 99.8%

Design 2 also experienced impressive performance improvement too. Frequency more than doubled to 16.21MHz from 8.010MHz. Besides that, Throughput and Throughput/gate also improved by approximately 100%.

6.8 FACTORS CAUSING DIFFERENCES BETWEEN ONE IMPLEMENTATION AND ANOTHER IMPLEMENTATION OF FPGA

There are many factors that contribute to the differences in performances between one implementation and another implementation on FPGA. Among them are as follows: -

- **Design**

  A design that was built carefully by considering all aspects of a target device would generally have a better performance. This is because the designer has selected a particular target device and tailored his design according to the architecture of the FPGA. This includes considering the number of slices, flip-flops, latches, Lookup Tables, etc. A general design

would generally not perform as well as to design that is tailored to a specific target device. Besides that, the smaller the number of input/output blocks used, the better the performance.

- **Target Device**

  Target device is also very important. As evident above, Spartan 3 performed 2 times better than Spartan 2. A bigger hardware resources, allows the device to optimize the design better. Usually when a device has taken up to 80% of the hardware resources, optimization becomes hard. Performance degradation would take place. This is evident in the design above (Spartan 2). In that design, 99% of the slices had been used. As a result optimization would not be very good and performance would degrade. As for the implementation on Spartan 3, 70% of the slices have been used. Spartan 3 has enough resources to further optimize on the design especially place and route.

- **Speed Grade**

  A higher speed grade would demonstrate a better performance difference. That means speed grade of 6 is better than speed grade 5 and so on. Theoretically it means, the FPGA is more suited for high speed application projects.

- **Architecture of FPGA**

  Architecture of the FPGA is also very important. Some architecture is meant for low speed applications, some architecture were meant for high speed processing, etc. Choosing the appropriate architecture and family type of FPGA is very important.

- **Synthesizer**

  Synthesizers too, play a very important role. In this project, I have used XST or Xilinx Synthesizing Tools. The best known synthesizing tool currently recommended is Synplify. These tools are capable of optimizing 30% more than most tools in the market.

Besides that, there is wide understanding among FPGA programmers that the Xilinx FPGAs could be executed at frequencies of 20 % more than that given by

synthesizers. That means 20 % more performance could be obtained than actually reported.

6.9 PERFORMANCE COMPARISON

One interesting point to consider is the performance of my designs as compared to other designs [5, 8, 10]. The table below shows the ranking of my designs.

Table 29: Performance Comparison With Other Implementations

Method	Designer	Device Technology	Gate Count	Throughput	Throughput per Gate
Hardware Evaluation	NSA	0.5um	945993	2.27Gbps	2403.32
Hardware Evaluation	Mitsubishi	0.35um	431857	394.08Mbps	912.52
FPGA	Kris Gaj	Xilinx XC4028XL	24800	90.9Mbps	3665
FPGA		Xilinx XVC1000	857560	1.59Gbps	1854.1
FPGA	Viktor Fischer	Altera Flex10K	41093.75	80.3Mbps	1954
FPGA (Design 1)	**Ananda Raja**	**Spartan2-XC2S200-5PQ208**	**26 318**	**34.9126Mbps**	**1326.5667**
FPGA (Design 2)	**Ananda Raja**	**Spartan2-XC2S200-5PQ208**	**32 616**	**56.9609Mbps**	**1746.4097**
***FPGA (Design 1)**	**Ananda Raja**	**Spartan 3 – XC3S400-5FG456**	**27 497**	**75.9842Mbps**	**2763.3644**
***FPGA (Design 2)**	**Ananda Raja**	**Spartan 3 – XC3S400-5FG456**	**33 044**	**115.3257Mbps**	**3490.0649**
ASIC	Yeong-Kang Lai, Liang-Gee Chen, Jian –Yi Lai, Tai-Ming Parng	.35um	35000	200Mbps	5714.28

*Benchmarking Purpose

Note: Design requirements between one implementation and another differ slightly eg. encryptor/decryptor, full keying/zero keying, ASIC/FPGA, etc

From the above diagram, the performance of both Design 1 and Design 2 that were implemented on **Spartan2- XC2S200-5PQ208** are comparable with other researchers. Both my designs were implemented with zero keying unlike some of the other designs. Furthermore, Design 1 and Design 2 could perform both encryption and decryption unlike some of the designs above. Therefore, the **table above is just for performance benchmarking. It would be incorrect to directly compare.**

Besides that, it is also important to note that, if both Design 1 and Design 2 were to be implemented with the following tools namely: -

- Higher resource and speed grade FPGA or ASIC
- Better synthesizing tool
- Latest - ISE6.3

a far higher performance could be achieved.

CHAPTER 7: CONCLUSION & RECOMMENDATIONS

7.1 CONCLUSION

Twofish is a 128-bit block cipher. It can work with variable key lengths: 128, 192 or 256 bits. In this report, only a version of 128-bit key length was discussed. Twofish has 6 main building blocks; Feistel Networks, whitening, S-boxes, MDS Matrices, Pseudo Hadamard Transforms and Key Schedule. Twofish is a 16 round Feistel network with a bijective F function, which corresponds to 8 cycles. The whitening technique employed substantially increases the difficulty of keysearch attacks against the remainder of the cipher. Twofish uses 4 different, bijective, key-dependent, 8-by-8 bit S-boxes. Twofish uses a single 4 by 4 MDS matrix over GF (2^8).This is one of the 2 main diffusion elements of Twofish. There is also Reed-Solomon code with the MDS property used in the key schedule; this doesn't add diffusion to the cipher but does add diffusion to the key schedule.) Besides that, Twofish also uses a 32 bit Pseudo Hadamard Transform to mix the outputs from its 2 parallel 32-bit g functions. Finally, Twofish needs a lot of key material, and has complicated key schedule. To facilitate analysis, the key schedule uses the same primitives as the round function. Except for 2 additional rotations, each pair of expanded key words is constructed by applying the Twofish round function (with key-dependent).

In this project, 2 different designs were implemented. The first design (Design 1) was implemented with minimum hardware resources usage, using a single F-Function (modified) and was optimized with reasonable latency, throughput and throughput per gate. As for the second design (Design 2) was implemented with reasonably minimum hardware resources using 4 units of F-Function(modified) of Design 1, minimum hardware resources usage, very small latency, very high throughput and very high throughput per gate. Furthermore, both Design 1 and Design 2 were implemented with zero keying and function as encryptor/decryptor. Both Design 1 and Design 2 were written using VHDL, simulated using ALDEC, synthesized using XILINX Synthesizing Tools, implemented using XILINX ISE6.2i implementation tools and download onto the Spartan 2 FPGA board using BEDLOAD utility program.

As a conclusion this Final Year Project is quite successful because all the objectives have been met successfully.

7.2 RECOMMENDATION

Since the implementation of this project has been very successful, it would be quite interesting to improve the project in other possible areas. Among the possible recommendations are as follows: -

- Perform a fast implementation Twofish Encryption Algorithm using mixed inner and outer round pipelining
- Implementing this algorithm for mobile communications.
- Implementing this algorithm for embedded systems.
- Propagation faults and their detection in a hardware implementation of Twofish Algorithm.

It would be quite interesting to implement the above recommendations as for the next level.

CHAPTER 8: REFERENCE

[1] B. Schneier," Applied Cryptography Second Edition", John Wiley & Sons, 1996.

[2] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, N.Ferguson, "TwoFish: a 128-bit block cipher",www.counterpane.com/twofish-paper.html

[3] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson, "The Twofish Encryption Algorithm", Wiley, 1999

[4] Pawel Chodowiec, Kris Gaj, "Implementation of the Twofish Cipher Using FPGA Devices", www.counterpane.com/twofishfpga.html

[5] Yeong-Kang Lai, Liang-Gee Chen, Jian-Yi Lai, and Tai-Ming Parng, "VLSI Architecture Design and Implementation for Twofish Block Cipher", proceedings of *IEEE International Symposium on Circuits & Systems (ISCAS'02)*, USA, May 26-29, 2002.

[6] Mark De Clercq and Vincent Levesque., "A VHDL Implementation of the Twofish Block Cipher", McGill University.

[7] Elbirt, A.J.; Yip, W.; Chetwynd, B.; Paar, C," An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists ", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, pp.545-557, Aug. 2001.

[8] T. Ichikawa, T. Kasuya and M. Matsui, "Hardware Evaluation of AES Finalists", in *The Third AES Candidate Conference*, Gaithersburg, MD, pp. 279-285, April 13-14, 2000.

[9] The National Institute of Standards and Technology (NIST), "Advanced Encryption Standard (AES)", *Federal Information Processing Standards Publication 197*, November 26, 2001.

[10] Viktor Fischer, "Realization of the Round 2 AES Candidates Using Altera FPGA", Technical Report, MICRONIC s.r.o.

APPENDIX A: MDS MATRIX

Let's assume we have an 8 bit number as defined below:

$S = S^7S^6S^5S^4S^3S^2S^1S^0$

This number is to be multiplied with an 8 bit number whereby the computation is to be performed in $GF(2^8)$ with a primitive polynomial:

$x^8+x^6+x^5+x^3+1$ (1 0110 1001)

The analysis has been done in Little Endian convention.

1) S multiply with 5B

Comments\bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	S0	S1	S2	S3	S4	S5	S6	S7								
x 5B	1	1	0	1	1	0	1	0								
	S0	S1	S2	S3	S4	S5	S6	S7								
		S0	S1	S2	S3	S4	S5	S6	S7							
			0	0	0	0	0	0	0	0						
				S0	S1	S2	S3	S4	S5	S6	S7					
					S0	S1	S2	S3	S4	S5	S6	S7				
						0	0	0	0	0	0	0	0			
							S0	S1	S2	S3	S4	S5	S6	S7		
								0	0	0	0	0	0	0	0	
Answer	S0	S0 XOR S1	S1 XOR S2	S0 XOR S2 XOR	S0 XOR S1 XOR	S1 XOR S2 XOR	S0 XOR S2 XOR	S1 XOR S3 XOR	S2 XOR S4 XOR	S3 XOR S5 XOR	S4 XOR S6 XOR	S5 XOR S7	S6	S7		

		S1	S2	S2 XOR S3 XOR S5	S1 XOR S3 XOR S4 XOR S6	S2 XOR S4 XOR S5 XOR S7	S2 XOR S3 XOR S6	S3 XOR S4 XOR S7	S4							
Primitive			1	0	0	1	0	1	1	0	1					
XOR			S4			S4		S4	S4		S4					
	S0	S0 XOR S1	S1 XOR S2 XOR S4	S0 XOR S2 XOR S3 XOR S5	S0 XOR S1 XOR S3 XOR S4 XOR S6	S1 XOR S2 XOR S5 XOR S7	S0 XOR S2 XOR S3 XOR S6	S1 XOR S3 XOR S7	S2	S3						
Primitive		1	0	0	1	0	1	1	0	1						
XOR S3		S3			S3		S3	S3		S3						
	S0	S0 XOR S1 XOR S3	S1 XOR S2 XOR S4	S0 XOR S2 XOR S3 XOR S5	S0 XOR S1 XOR S4 XOR S6	S1 XOR S2 XOR S5 XOR S7	S0 XOR S2 XOR S6	S1 XOR S7	S2							
Primitive	1	0	0	1	0	1	1	0	1							
XOR S2	S2			S2		S2	S2		S2							
Final Answer (Recovered)	S0 XOR	S0 XOR	S1 XOR	S0 XOR	S0 XOR	S1 XOR	S0 XOR	S1 XOR								

				S3	S3 XOR S4	S4 XOR S5	S3 XOR S5 XOR S6	S4 XOR S6 XOR S7	S5 XOR S7	S6	S7				
Primitive						1	0	0	1	0	1	1	0	1		
XOR S7						S7			S7		S7	S7		S7		
	S0	S0 XOR S1	S1 XOR S2	S0 XOR S2 XOR S3	S0 XOR S1 XOR S3 XOR S4	S1 XOR S2 XOR S4 XOR S5 XOR S7	S0 XOR S2 XOR S3 XOR S5 XOR S6	S1 XOR S3 XOR S4 XOR S6 XOR S7	S2 XOR S4 XOR S5	S3 XOR S5 XOR S6	S4 XOR S6	S5	S6			
Primitive					1	0	0	1	0	1	1	0	1			
XOR S6					S6			S6		S6	S6		S6			
	S0	S0 XOR S1	S1 XOR S2	S0 XOR S2 XOR S3	S0 XOR S1 XOR S3 XOR S4 XOR S6	S1 XOR S2 XOR S4 XOR S5 XOR S7	S0 XOR S2 XOR S3 XOR S5 XOR S6	S1 XOR S3 XOR S4 XOR S6 XOR S7	S2 XOR S4 XOR S5	S3 XOR S5 XOR S6	S4 XOR S6	S5	S6			
Primitive				1	0	0	1	0	1	1	0	1				
XOR S5				S5			S5		S5	S5		S5				
	S0	S0 XOR	S1 XOR	S0 XOR	S0 XOR	S1 XOR	S0 XOR	S1 XOR	S2 XOR	S3	S4					

		S2	S1 XOR S3	S2 XOR S4	S3 XOR S5	S1 XOR S4 XOR S6	S5 XOR S7	S6	S7							

2) S multiply with EF

Comments\bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	S0	S1	S2	S3	S4	S5	S6	S7								
x EF	1	1	1	1	0	1	1	1								
	S0	S1 S0	S2 S1 S0	S3 S2 S1 S0	S4 S3 S2 S1 0	S5 S4 S3 S2 0 S0	S6 S5 S4 S3 0 S1 S0	S7 S6 S5 S4 0 S2 S1 S0	S7 S6 S5 S4 0 S3 S2 S1	S7 S6 S5 0 S4 S3 S2	S7 S6 0 S5 S4 S3	0 S6 S5 S4 S3	S7 S6 S5 S4	S7 S6 S5	S7 S6	
Answer	S0	S0 XOR S1`	S0 XOR S1 XOR S2	S0 XOR S1 XOR S2 XOR S3	S1 XOR S2 XOR S3 XOR S4	S0 XOR S2 XOR S3 XOR S4 XOR S5	S0 XOR S1 XOR S3 XOR S4 XOR S5 XOR S6	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S3 XOR S4 XOR S5 XOR S6 XOR S7	S4 XOR S5 XOR S6 XOR S7	S5 XOR S6 XOR S7	S6 XOR S7	S7	
Primitive							1	0	0	1	0	1	1	0	1	
XOR S7							S7			S7		S7	S7		S7	

	S0	S0 XOR S1`	S0 XOR S1 XOR S2	S0 XOR S1 XOR S2 XOR S3	S0 XOR S1 XOR S2 XOR S3 XOR S4	S1 XOR S2 XOR S3 XOR S4 XOR S5	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6	S0 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S3 XOR S4 XOR S5 XOR S6 XOR S7	S4 XOR S5 XOR S6 XOR S7	S5 XOR S6	S6 XOR S7		
Primitive								1	0	0	1	0	0	1	1	0	1		
XOR (S6 XOR S7)								S6 XOR S7			S6 XOR S7		S6 XOR S7	S6 XOR S7	S6 XOR S7		S6 XOR S7		
	S0	S0 XOR S1`	S0 XOR S1 XOR S2	S0 XOR S1 XOR S2 XOR S3	S0 XOR S1 XOR S2 XOR S3 XOR S4	S1 XOR S2 XOR S3 XOR S4 XOR S5	S0 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6	S0 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S3 XOR S4 XOR S5 XOR S6 XOR S7	S4 XOR S5 XOR S6 XOR S7	S5 XOR S6			
Primitive								1	0	0	1	0	1	1	0	1			

XOR (S5 XOR S6)						S5 XOR S6			S5 XOR S6		S5 XOR S6	S5 XOR S6		S5 XOR S6			
	S0	S0 XOR S1`	S0 XOR S1 XOR S2	S0 XOR S1 XOR S2 XOR S3	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6	S0 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S4 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S3 XOR S5 XOR S6 XOR S7	S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S3 XOR S4 XOR S5 XOR S6 XOR S7	S4 XOR S5 XOR S6 XOR S7	S5 XOR S6			
Primitive				1	0	0	1	0	1	1	0	1					
XOR (S4 XOR S5)				S4 XOR S5			S4 XOR S5		S4 XOR S5	S4 XOR S5		S4 XOR S5					
	S0	S0 XOR S1`	S0 XOR S1 XOR S2	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S5	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6	S0 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR	S0 XOR S1 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S4 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S3 XOR S5 XOR S6 XOR S7	S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S3 XOR S4 XOR S5 XOR S6 XOR S7				

							S7										
Primitive			1	0	0	1	0	1	1	0	1						
XOR (S3 XOR S4)			S3 XOR S4			S3 XOR S4		S3 XOR S4	S3 XOR S4		S3 XOR S4						
	S0	S0 XOR S1`	S0 XOR S1 XOR S2 XOR S3 XOR S4	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S5	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6	S0 XOR S2 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S3 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S3 XOR S7	S1 XOR S2	S2 XOR S3							
Primitive		1	0	0	1	0	1	1	0	1							
XOR (S2 XOR S3)		S2 XOR S3			S2 XOR S3		S2 XOR S3	S2 XOR S3		S2 XOR S3							
	S0	S0 XOR S1` XOR S2 XOR S3	S0 XOR S1 XOR S2 XOR S3 XOR S4	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S5	S1 XOR S4 XOR S5 XOR S6	S0 XOR S2 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S7	S1 XOR S2							

Primitive	1	0	0	1	0	1	1	0	1							
XOR (S1 XOR S2)	S1 XOR S2			S1 XOR S2		S1 XOR S2	S1 XOR S2		S1 XOR S2							
Final Answer (Recovered)	S0 XOR S1 XOR S2	S0 XOR S1 XOR S2 XOR S3	S0 XOR S1 XOR S2 XOR S3 XOR S4	S0 XOR S3 XOR S4 XOR S5	S1 XOR S4 XOR S5 XOR S6	S0 XOR S1 XOR S5 XOR S6 XOR S7	S0 XOR S6 XOR S7	S0 XOR S1 XOR S7								

APPENDIX B: REED SOLOMON MATRIX

Let's assume we have an 8 bit number as defined below:

$S = S^7S^6S^5S^4S^3S^2S^1S^0$

This number is to be multiplied with an 8 bit number whereby the computation is to be performed in $GF(2^8)$ with a primitive polynomial:

$x^8+x^6+x^3+x^2+1$ (1 0100 1101)

The analysis has been done in Little Endian convention.

1) S multiply with A4

Comments\bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	S0	S1	S2	S3	S4	S5	S6	S7								
x A4	0	0	1	0	0	1	0	1								
	0	0	0	0	0	0	0	0								
		0	0	0	0	0	0	0	0							
			S0	S1	S2	S3	S4	S5	S6	S7						
				0	0	0	0	0	0	0	0					
					0	0	0	0	0	0	0	0				
						S0	S1	S2	S3	S4	S5	S6	S7			
							0	0	0	0	0	0	0			
								S0	S1	S2	S3	S4	S5	S6	S7	

Answer	0	0	S0	S1	S2	S0 XOR S3	S1 XOR S4	S0 XOR S2 XOR S5	S1 XOR S3 XOR S6	S2 XOR S4 XOR S7	S3 OR S5	S4 OR S6	S5 XOR S7	S6	S7	
Primitive							1	0	1	1	0	0	1	0	1	
X0R S7							S7		S7	S7			S7		S7	
	0	0	S0	S1	S2	S0 XOR S3	S1 XOR S4	S0 XOR S2 XOR S7	S1 XOR S3 XOR S5	S2 XOR S4 XOR S6	S3 XOR S5 XOR S7	S4 XOR S6	S5	S6		
Primitive						1	0	1	1	0	0	1	0	1		
X0R S6						S6		S6	S6			S6		S6		
	0	0	S0	S1	S2	S0 XOR S3	S1 XOR S4	S0 XOR S2 XOR S6	S1 XOR S3 XOR S5	S2 XOR S4 XOR S7	S3 XOR S5	S4	S5			

							S6	S7	S5 XOR S6	S7						
Primitive						1	0	1	1	0	0	1	0	1		
X0R S5						S5		S5	S5			S5		S5		
	0	0	S0	S1	S2 XOR S5	S0 XOR S3	S1 XOR S4 XOR S6	S0 XOR S2 XOR S5 XOR S7	S1 XOR S3 XOR S6	S0 XOR S2 XOR S4 XOR S7	S1 XOR S3 XOR S5	S2 XOR S4 XOR S6	S3 XOR S5	S4		
Primitive					1	0	1	1	0	0	1	0	1			
X0R S4					S4		S4	S4			S4		S4			
	0	0	S0	S1 XOR S4	S2 XOR S5	S0 XOR S3	S1 XOR S4 XOR S6	S0 XOR S2 XOR S5 XOR S7	S1 XOR S3 XOR S6	S0 XOR S2 XOR S4 XOR S7	S1 XOR S3 XOR S5	S2 XOR S4 XOR S6	S3 XOR S5	S4		

						S6										
Primitive			1	0	1	1	0	0	1	0	1					
X0R S3			S3		S3	S3			S3		S3					
	0	0	S0	S1	S2	S0	S1	S0	S1	S2						
			X0R	X0R	X0R	X0R	X0R	X0R	X0R							
			S3	S4	S3	S4	S5	S2	S7							
					X0R	X0R	X0R	X0R								
					S5	S6	S7	S6								
Primitive		1	0	1	1	0	0	1	0	1						
X0R S2		S2		S2	S2			S2		S2						
	0	S2	S0	S1	S3	S0	S1	S0	S1							
			X0R	X0R	X0R	X0R	X0R	X0R	X0R							
			S3	S2	S5	S4	S5	S6	S7							
				X0R		X0R	X0R									
				S4		S6	S7									
Primitive	1	0	1	1	0	0	1	0	1							
X0R (S1 X0R S7)	S1		S1	S1			S1		S1							
	X0R		X0R	X0R			X0R		X0R							
	S7		S7	S7			S7		S7							
FINAL ANSWER	S1	S2	S0	S2	S3	S0	S5	S0								

(REDUCED)	X0R		X0R	X0R	X0R	X0R		X0R								
	S7		S1	S4	S5	S4		S6								
			X0R	X0R		X0R										
			S3	S7		S6										
			X0R													
			S7													

2) S multiply with 55

Comments\bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	S0	S1	S2	S3	S4	S5	S6	S7								
x 55	1	0	1	0	1	0	1	0								
	S0	S1	S2	S3	S4	S5	S6	S7								
		0	0	0	0	0	0	0	0							
			S0	S1	S2	S3	S4	S5	S6	S7						
				0	0	0	0	0	0	0	0					
					S0	S1	S2	S3	S4	S5	S6	S7				
						0	0	0	0	0	0	0	0			
							S0	S1	S2	S3	S4	S5	S6	S7		
								0	0	0	0	0	0	0	0	
Answer	S0	S1	S0	S1	S0	S1	S0	S1	S2	S3	S4	S5	S6	S7	0	
			X0R	X0R	X0R	X0R	X0R	X0R	X0R	X0R	X0R	X0R				
			S2	S3	S2	S3	S2	S3	S4	S5	S6	S7				
					X0R	X0R	X0R	X0R	X0R	X0R						
					S4	S5	S4	S5	S6	S7						
							X0R	X0R								
							S6	S7								

Primitive						1	0	1	1	0	0	1	0	1		
X0R S7						S7		S7	S7			S7		S7		
	S0	S1	S0	S1	S0	S1	S0	S1	S2	S3	S4	S5	S6			
			X0R	X0R	X0R	X0R	X0R	X0R	X0R	X0R	X0R					
			S2	S3	S2	S3	S2	S3	S4	S5	S6					
					X0R	X0R	X0R	X0R	X0R	X0R						
					S4	S5	S4	S5	S6	S7						
						X0R	X0R		X0R	X0R						
						S7	S6		S7							
Primitive					1	0	1	1	0	0	1	0	1			
X0R S6					S6		S6	S6			S6		S6			
	S0	S1	S0	S1	S0	S1	S0	S1	S2	S3	S4	S5				
			X0R	X0R	X0R	X0R	X0R	X0R	X0R	X0R						
			S2	S3	S2	S3	S2	S3	S4	S5						
					0R	X0R	X0R	X0R	X0R	X0R						
					S4	S5	S4	S5	S6	S7						
					X0R	X0R		X0R	X0R							
					S6	S7		S6	S7							
Primitive				1	0	1	1	0	0	1	0	1				
X0R S5				S5		S5	S5			S5		S5				
	S0	S1	S0	S1	S0	S1	S0	S1	S2	S3	S4					
			X0R	X0R	X0R	X0R	X0R	X0R	X0R	X0R						

			S2	S3 XOR S5	S2 0R S4 XOR S6	S3 XOR S7	S2 XOR S4	S3 XOR S5 XOR S6	S4 XOR S6 XOR S7	S7						
Primitive			1	0	1	1	0	0	1	0	1					
X0R S4			S4		S4	S4			S4		S4					
	S0	S1	S0 XOR S2 XOR S4	S1 XOR S3 XOR S5	S0 XOR S2 XOR S6	S1 XOR S3 XOR S4 XOR S7	S0 XOR S2 XOR S4 XOR S5 XOR S7	S1 XOR S3 XOR S5 XOR S6	S2 XOR S4 XOR S5 XOR S7	S3 XOR S7						
Primitive		1	0	1	1	0	0	1	0	1						
X0R (S3 X0R S7)		S3 X0R S7		S3 X0R S7	S3 X0R S7			S3 X0R S7		S3 X0R S7						
	S0	S1 X0R S3 X0R	S0 X0R S2 X0R	S1 X0R S2 X0R	S0 X0R S3 X0R	S1 X0R S2 X0R	S0 X0R S1 X0R	S1 X0R S2 X0R	S2 X0R S3 X0R							

		S7	S4	S7	X0R S3 X0R S6 X0R S7	S4 X0R S7	S4 X0R S5	S6 X0R S7	S7							
Primitive	1	0	1	1	0	0	1	0	1							
X0R (S2 X0R S6 X0R S7)	S2 X0R S6 X0R S7		S2 X0R S6 X0R S7	S2 X0R S6 X0R S7			S2 X0R S6 X0R S7		S2 X0R S6 X0R S7							
FINAL ANSWER (REDUCED)	S0 X0R S2 X0R S6 X0R S7	S1 X0R S3 X0R S7	S0 X0R S4 X0R S6 X0R S7	S1 X0R S2 X0R S5 X0R S6 X0R S7	S0 X0R S2 X0R S3 X0R S6 X0R S7	S1 X0R S3 X0R S4 X0R S5 X0R S6 X0R S7	S0 X0R S4 X0R S5 X0R S6 X0R S7	S1 X0R S5 X0R S6 X0R S7								

3) S multiply with 87

Comments\bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	S0	S1	S2	S3	S4	S5	S6	S7								
x 87	1	1	1	0	0	0	0	1								
	S0	S1 S0	S2 S1 S0	S3 S2 S1 0	S4 S3 S2 0 0	S5 S4 S3 0 0 0	S6 S5 S4 0 0 0 0	S7 S6 S5 S0	S7 S6 S5 S1	S7 S6 S2	0 0 S3	0 0 S4	0 0 S5	0 0 S6	S7	
Answer	S0	S0 XOR S1	S0 XOR S1 XOR S2	S1 XOR S2 XOR S3	S2 XOR S3 XOR S4	S3 XOR S4 XOR S5	S4 XOR S5 XOR S6	S0 XOR S5 XOR S6 XOR S7	S1 XOR S6 XOR S7	S2 XOR S7	S3	S4	S5	S6	S7	
Primitive							1	0	1	1	0	0	1	0	1	
XOR S7							S7		S7	S7			S7		S7	
	S0	S0 XOR S1	S0 XOR S1 XOR S2	S1 XOR S2 XOR S3	S2 XOR S3 XOR S4	S3 XOR S4 XOR S5	S4 XOR S5 XOR S6 XOR S7	S0 XOR S5 XOR S6 XOR S7	S1 XOR S6	S2	S3	S4	S5 XOR S7	S6		

Primitive						1	0	1	1	0	0	1	0	1		
XOR S6						S6		S6	S6			S6		S6		
	S0	S0 XOR S1	S0 XOR S1 XOR S2	S1 XOR S2 XOR S3	S2 XOR S3 XOR S4	S3 XOR S4 XOR S5 XOR S6	S4 XOR S5 XOR S6 XOR S7	S0 XOR S6 XOR S7	S1	S2	S3	S4 XOR S6	S5	S6 XOR S7		
Primitive					1	0	1	1	0	0	1	0	1			
XOR (S5 XOR S7)					S5 XOR S7		S5 XOR S7	S5 XOR S7			S5 XOR S7		S5 XOR S7			
	S0	S0 XOR S1	S0 XOR S1 XOR S2	S1 XOR S2 XOR S3	S2 XOR S3 XOR S4 XOR S5 XOR S7	S3 XOR S4 XOR S5 XOR S6	S4 XOR S6	S0	S1	S2	S3 XOR S5 XOR S7	S4 XOR S6				
Primitive				1	0	1	1	0	0	1	0	1				
XOR (S4 XOR S6)				S4 XOR S6		S4 XOR S6	S4 XOR S6			S4 XOR S6		S4 XOR S6				
	S0	S0 XOR S1	S0 XOR S1 XOR S2	S1 XOR S2 XOR S3	S2 XOR S3 XOR S5	S3 XOR S5		S0	S1	S2 XOR S4 XOR	S3 XOR S5 XOR					

			S2	S3 XOR S4 XOR S6	S4 XOR S5 XOR S7					S6	S7						
Primitive			1	0	1	1	0	0	1	0	1						
XOR (S3 XOR S5 XOR S7)			S3 XOR S5 XOR S7		S3 XOR S5 XOR S7	S3 XOR S5 XOR S7			S3 XOR S5 XOR S7		S3 XOR S5 XOR S7						
	S0	S0 XOR S1	S0 XOR S1 XOR S2 XOR S3 XOR S5 XOR S7	S1 XOR S2 XOR S3 XOR S4 XOR S6	S2 XOR S4	S7		S0	S1 XOR S3 XOR S5 XOR S7	S2 XOR S4 XOR S6							
Primitive		1	0	1	1	0	0	1	0	1							
XOR (S2 XOR S4 XOR S7)		S2 XOR S4 XOR S6		S2 XOR S4 XOR S6	S2 XOR S4 XOR S6			S2 XOR S4 XOR S6		S2 XOR S4 XOR S6							
	S0	S0 XOR S1	S0 XOR S1	S1 XOR S3	S6	S7		S0 XOR S2	S1 XOR S3								

		XOR S2 XOR S4 XOR S6	XOR S2 XOR S3 XOR S5 XOR S7					XOR S4 XOR S6	XOR S5 XOR S7								
Primitive	1	0	1	1	0	0	1	0	1								
XOR (S1 XOR S3 XOR S5 XOR S7)	S1 XOR S3 XOR S5 XOR S7		S1 XOR S3 XOR S5 XOR S7	S1 XOR S3 XOR S5 XOR S7			S1 XOR S3 XOR S5 XOR S7		S1 XOR S3 XOR S5 XOR S7								
FINAL ANSWER (REDUCED)	S0 XOR S1 XOR S3 XOR S5 XOR S7	S0 XOR S1 XOR S2 XOR S4 XOR S6	S0 XOR S2 XOR S7	S5 XOR S7	S6	S7	S1 XOR S3 XOR S5 XOR S7	S0 XOR S2 XOR S4 XOR S6									

4) S multiply with 5A

Comments\bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	S0	S1	S2	S3	S4	S5	S6	S7								
x 5A	0	1	0	1	1	0	1	0								
	0	0	0	0	0	0	0	0								
		S0	S1	S2	S3	S4	S5	S6	S7							
			0	0	0	0	0	0	0	0						
				S0	S1	S2	S3	S4	S5	S6	S7					
					S0	S1	S2	S3	S4	S5	S6	S7				
						0	0	0	0	0	0	0	0			
							S0	S1	S2	S3	S4	S5	S6	S7		
								0	0	0	0	0	0	0	0	
Answer		S0	S1	S0 XOR S2	S0 XOR S1 XOR S3	S1 XOR S2 XOR S4	S0 XOR S2 XOR S3	S1 XOR S3 XOR S4 XOR S5	S2 XOR S4 XOR S5 XOR S6	S3 XOR S5 XOR S6 XOR S7	S4 XOR S6 XOR S7	S5 XOR S7	S6	S7		
Primitive						1	0	1	1	0	0	1	0	1		
XOR S7						S7		S7	S7			S7		S7		
		S0	S1	S0 XOR S2	S0 XOR S1 XOR S3	S1 XOR S2 XOR S4 XOR S7	S0 XOR S2 XOR S3 XOR S5	S1 XOR S3 XOR S4 XOR S6	S2 XOR S4 XOR S5 XOR S7	S3 XOR S5 XOR S6	S4 XOR S6 XOR S7	S5	S6			

								XOR S7									
Primitive					1	0	1	1	0	0	1	0	1				
XOR S6					S6		S6	S6			S6		S6				
		S0	S1	S0 XOR S2	S0 XOR S1 XOR S3 XOR S6	S1 XOR S2 XOR S4 XOR S7	S0 XOR S2 XOR S3 XOR S5 XOR S6	S1 XOR S3 XOR S4 XOR S6 XOR S7	S2 XOR S4 XOR S5 XOR S7	S3 XOR S5 XOR S6	S4 XOR S7	S5					
Primitive					1	0	1	1	0	0	1	0	1				
XOR S5					S5		S5	S5			S5		S5				
		S0	S1	S0 XOR S2	S0 XOR S1 XOR S2 XOR S3 XOR S5	S1 XOR S2 XOR S3 XOR S4 XOR S6	S0 XOR S2 XOR S3 XOR S4 XOR S5	S1 XOR S2 XOR S3 XOR S4 XOR S6 XOR S7	S2 XOR S3 XOR S4 XOR S5	S3 XOR S4 XOR S6	S4 XOR S7						
Primitive			1	0	1	1	0	0	1	0	1						
XOR (S4 XOR S7)			S4 XOR S7		S4 XOR S7	S4 XOR S7			S4 XOR S7		S4 XOR S7						
		S0	S1 XOR S4	S0 XOR S2	S0 XOR S1 S2	S1 XOR S2	S0 XOR S2	S1 XOR S3	S2 XOR S5	S3 XOR S6							

			XOR S7	XOR S5	XOR S3 XOR S4 XOR S6 XOR S7	XOR S5	XOR S3 XOR S6	XOR S4 XOR S7	XOR S7							
Primitive		1	0	1	1	0	0	1	0	1						
XOR (S3 XOR S6)		S3 XOR S6		S3 XOR S6	S3 XOR S6			S3 XOR S6		S3 XOR S6						
		SO XOR S3 XOR S6	S1 XOR S4 XOR S7	S0 XOR S2 XOR S3 XOR X5 XOR S6	S0 XOR S1 XOR S4 XOR S7	S1 XOR S2 XOR S5	S0 XOR S2 XOR S3 XOR S6	S1 XOR S4 XOR S6 XOR S7	S2 XOR S5 XOR S7							
Primitive	1	0	1	1	0	0	1	0	1							
XOR (S2 XOR S5 XOR S7)	S2 XOR S5 XOR S7		S2 XOR S5 XOR S7	S2 XOR S5 XOR S7			S2 XOR S5 XOR S7		S2 XOR S5 XOR S7							
FINAL ANSWER (REDUCED)	S2 XOR S5 XOR	SO XOR S3 XOR	S1 XOR S2 XOR	S0 XOR S3 XOR	S0 XOR S1 XOR	S1 XOR S2 XOR	S0 XOR S3 XOR	S1 XOR S4 XOR								

	S7	S6	S4 XOR S5	S6 XOR S7	S4 XOR S7	S5	S5 XOR S6 XOR S7	S6 XOR S7								

5) S multiply with 58

Comments\bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	S0	S1	S2	S3	S4	S5	S6	S7								
x 58	0	0	0	1	1	0	1	0								
	0	0	0	0	0	0	0	0								
		0	0	0	0	0	0	0	0							
			0	0	0	0	0	0	0	0	S7					
				S0	S1	S2	S3	S4	S5	S5	S6	S7				
					S0	S1	S2	S3	S4	S5	S6	S7				
						0	0	0	0	0	0	0	0			
							S0	S1	S2	S3	S4	S5	S6	S7		
								0	0	0	0	0	0	0	0	
Answer	0	0	0	S0	S0 XOR S1	S1 XOR S2	S0 XOR S2 XOR S3	S1 XOR S2 XOR S3 XOR S4	S2 XOR S3 XOR S4 XOR S5	S3 XOR S4 XOR S5 XOR S6	S4 XOR S5 XOR S6 XOR S7	S5 XOR S6 XOR S7	S6 XOR S7	S7		
Primitive						1	0	1	1	0	0	1	0	1		
XOR S7						S7		S7	S7			S7		S7		
	0	0	0	S0	S0 XOR S1	S1 XOR S2 XOR S7	S0 XOR S2 XOR S3	S1 XOR S2 XOR S3 XOR S4	S2 XOR S3 XOR S4 XOR S5	S3 XOR S4 XOR S5 XOR S6	S4 XOR S5 XOR S6 XOR S7	S5 XOR S6 XOR S7	S6 XOR S7			

									S7	S7						
Primitive					1	0	1	1	0	0	1	0	1			
XOR S6					S6		S6	S6			S6		S6			
	0	0	0	S0	S0 XOR S1 XOR S6	S1 XOR S2 XOR S7	S0 XOR S2 XOR S3 XOR S6	S1 XOR S2 XOR S3 XOR S4 XOR S6 XOR S7	S2 XOR S3 XOR S4 XOR S5 XOR S7	S3 XOR S4 XOR S5 XOR S6	S4 XOR S5 XOR S7	S5				
Primitive				1	0	1	1	0	0	1	0	1				
XOR S5				S5		S5	S5			S5		S5				
	0	0	0	S0 XOR S5	S0 XOR S1 XOR S5	S1 XOR S2 XOR S6	S0 XOR S2 XOR S3 XOR S5 XOR S6	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S7	S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S3 XOR S4 XOR S5 XOR S6 XOR S7	S4 XOR S5 XOR S6 XOR S7	S6 XOR S7				
Primitive			1	0	1	1	0	0	1	0	1					
XOR (S4 XOR S7)			S4 XOR S7		S4 XOR S7	S4 XOR S7			S4 XOR S7		S4 XOR S7					
			S4 XOR S7	S0 XOR S5	S0 XOR S1 XOR	S1 XOR S2 XOR	S0 XOR S2 XOR	S1 XOR S2 XOR	S2 XOR S3 XOR S5	S3 XOR S6						

					S4 XOR S6 XOR S7	S4 XOR S5	S3 XOR S5 XOR S6	S4 XOR S6 XOR S7								
Primitive		1	0	1	1	0	0	1	0	1						
XOR (S3 XOR S6)		S3 XOR S6		S3 XOR S6	S3 XOR S6			S3 XOR S6		S3 XOR S6						
		S3 XOR S6	S4 XOR S7	S0 XOR S3 XOR S5 XOR S6	S0 XOR S1 XOR S3 XOR S4 XOR S7	S1 XOR S2 XOR S4 XOR S5	S0 XOR S2 XOR S3 XOR S5 XOR S6	S1 XOR S4 XOR S7	S2 XOR S5							
Primitive	1	0	1	1	0	0	1	0	1							
XOR (S2 XOR S5)	S2 XOR S5		S2 XOR S5	S2 XOR S5			S2 XOR S5		S2 XOR S5							
Final Answer (Recovered)	S2 XOR S5	S3 XOR S6	S2 XOR S4 XOR S5 XOR S7	S0 XOR S2 XOR S3 XOR S6	S0 XOR S1 XOR S3 XOR S4	S1 XOR S2 XOR S4 XOR S5	S0 XOR S3 XOR S6	S1 XOR S4 XOR S7								

								XOR S7								

6) S multiply with DB

Comments\bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	S0	S1	S2	S3	S4	S5	S6	S7								
x DB	1	1	0	1	1	0	1	1								
	S0	S1 S0	S2 S1 0	S3 S2 0 S0	S4 S3 0 S1 S0	S5 S4 0 S2 S1 0	S6 S5 0 S3 S2 0 S0	S7 S6 0 S4 S3 0 S1 S0	S7 0 S5 S4 0 S2 S1	0 S6 S5 0 S3 S2	S7 S6 0 S4 S3	S7 0 S5 S4	0 S6 S5	0 S7 S6	S7 S6	S7
Answer	S0	S0 XOR S1	S1 XOR S2	S0 XOR S2 XOR S3	S0 XO S1	S1 XOR S2	S0 XOR S2 XOR S3	S0 XOR S1 XOR S3 XOR S4	S1 XOR S2 XOR S4 XOR S5	S2 XOR S3 XR S5 XOR S6	S3 XOR S4 XOR S5 XOR S7	S4 XOR S5 XOR S6	S5 XOR S6	S6 XOR S7	S7	
Primitive							1	0	1	1	0	0	1	0	1	
XOR S7							S7		S7	S7			S7		S7	
	S0	S0 XOR S1	S1 XOR S2	S0 XOR S2	S0 XO S1	S1 XOR S2	S0 XOR S2	S0 XOR S1	S1 XOR S2	S2 XOR S3	S3 XOR S4	S4 XOR S5	S5 XOR S6	S6 XOR S7		

				XOR S3	XOR S3 XOR S4	XOR S4 XOR S5	XOR S3 XOR S5 XOR S6	XOR S3 XOR S4 XOR S6 XOR S7	XOR S4 XOR S5 XOR S6 XOR S7	XOR S5 XOR S6 XOR S7	XOR S6 XOR S7	XOR S7	XOR S7			
Primitive							1	0	1	1	0	0	1	0	1	
XOR (S6 XOR S7)							S6 XOR S7		S6 XOR S7	S6 XOR S7			S6 XOR S7		S6 XOR S7	
	S0	S0 XOR S1	S1 XOR S2	S0 XOR S2 XOR S3	S0 XO S1	S0 XOR S2	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S4 XOR S5 XOR S6 XOR S7	S2 XOR S3 XOR S5 XOR S6 XOR S7	S3 XOR S4 XOR S6 XOR S7	S4 XOR S5 XOR S7	S5 XOR S6 XOR S7	S5 XOR S6 XOR S7	
Primitive								1	0	1	1	0	0	1	0	1
XOR (S5 XOR S6 XOR S7)								S5 XOR S6 XOR S7		S5 XOR S6 XOR S7	S5 XOR S6 XOR S7			S5 XOR S6 XOR S7		S5 XOR S6 XOR S7
	S0	S0 XOR	S1 XOR	S0 XOR	S0 XOR	S1 XOR	S0 XOR	S0 XOR	S1 XOR	S0 XOR	S0 XOR	S1 XOR	S2 XOR	S3 XOR	S4 XOR	

		S1	S2	S2 XOR S3	S1 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S2 XOR S4 XOR S5 XOR S6 XOR S7	S2 XOR S3	S1 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S2 XOR S4 XOR S5 XOR S6 XOR S7	S3 XOR S5 XOR S6 XOR S7	S4 XOR S5	S5 XOR S6					
Primitive				1	0	1	1	0	0	1	0	1					
XOR (S4 XOR S5 XOR S6)				S4 XOR S5 XOR S6		S4 XOR S5 XOR S6	S4 XOR S5 XOR S6			S4 XOR S5 XOR S6		S4 XOR S5 XOR S6					
	S0	S0 XOR S1	S1 XOR S2	S0 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6	S0 XOR S1 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S7	S0 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6	S0 XOR S1 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S4 XOR S5 XOR S6 XOR S7	S2 XOR S3 XOR S4 XOR S5 XOR S7	S3 XOR S4 XOR S5						
Primitive			1	0	1	1	0	0	1	0	1						

XOR (S3 XOR S4 XOR S5)			S3 XOR S4 XOR S5		S3 XOR S4 XOR S5	S3 XOR S4 XOR S5			S3 XOR S4 XOR S5		S3 XOR S4 XOR S5					
	S0	S0 XOR S1	S1 XOR S2 XOR S3 XOR S4 XOR S5	S0 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6	S0 XOR S1 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7						
Primitive		1	0	1	1	0	0	1	0	1							
XOR (S2 XOR S3 XOR S4 XOR S7)		S2 XOR S3 XOR S4 XOR S7		S2 XOR S3 XOR S4 XOR S7	S2 XOR S3 XOR S4 XOR S7			S2 XOR S3 XOR S4 XOR S7		S2 XOR S3 XOR S4 XOR S7							
	S0	S0 XOR S1 XOR S2 XOR	S1 XOR S2 XOR S3 XOR	S0 XOR S5 XOR S6 XOR	S0 XOR S1 XOR S2 XOR	S1 XOR S2 XOR S3 XOR	S0 XOR S2 XOR S3 XOR	S0 XOR S1 XOR S2 XOR	S1 XOR S2 XOR S3 XOR							

		S3 XOR S4 XOR S5	S4 XOR S5	S7	S3 XOR S4 XOR S6	S4 XOR S5 XOR S7	S4 XOR S5 XOR S6	S5 XOR S6	S6 XOR S7							
Primitive	1	0	1	1	0	0	1	0	1							
XOR (S1 XOR S2 XOR S3 XOR S6 XOR S7)	S1 XOR S2 XOR S3 XOR S6 XOR S7		S1 XOR S2 XOR S3 XOR S6 XOR S7	S1 XOR S2 XOR S3 XOR S6 XOR S7			S1 XOR S2 XOR S3 XOR S6 XOR S7		S1 XOR S2 XOR S3 XOR S6 XOR S7							
Final Answer (Recovered)	S0 XOR S1 XOR S2 XOR S3 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S7	S4 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S3 XOR S5 XOR S7	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S7	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6	S0 XOR S1 XOR S2 XOR S3 XOR S5 XOR S6								

7) S multiply with 9E

Comments\bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	S0	S1	S2	S3	S4	S5	S6	S7								
x 9E	0	1	1	1	1	0	0	1								
	0	0 S0	0 S1 S0	0 S2 S1 S0	0 S3 S2 S1 S0	0 S4 S3 S2 S1 0	0 S5 S4 S3 S2 0 0	0 S6 S5 S4 S3 0 0 S0	S7 S6 S5 S4 0 0 S1	S7 S6 S5 0 0 S2	S7 S6 0 0 S3	S7 0 0 S4	0 0 S5	0 0 S6	S7	
Answer		S0	S0 XOR S1	S0 XOR S1 XOR S2	S0 XOR S1 XOR S2 XOR S3	S1 XOR S2 XOR S3 XOR S4	S2 XOR S3 XOR S4 XOR S5	S0 XOR S3 XOR S4 XOR S5 XOR S6	S1 XOR S4 XOR S5 XOR S6 XOR S7	S2 XOR S5 XOR S6 XOR S7	S3 XOR S6 XOR S7	S4 XOR S7	S5	S6	S7	
Primitive							1	0	1	1	0	0	1	0	1	
XOR S7							S7		S7	S7			S7		S7	
		S0	S0 XOR S1	S0 XOR S1 XOR S2	S0 XOR S1 XOR S2	S1 XOR S2 XOR S3	S2 XOR S3 XOR S4	S0 XOR S3 XOR S4	S1 XOR S4 XOR S5	S2 XOR S5 XOR S6	S3 XOR S6 XOR S7	S4 XOR S7	S5 XOR S7	S6		

					XOR S3	XOR S4	XOR S5 XOR S7	XOR S6 XOR S6	XOR S6								
Primitive						1	0	1	1	0	0	1	0	1			
XOR S6						S6		S6	S6			S6		S6			
		S0	S0 XOR S1	S0 XOR S1 XOR S2	S0 XOR S2 XOR S3	S1 XOR S2 XOR S3 XOR S4	S2 XOR S3 XOR S4 XOR S5	S0 XOR S3 XOR S4 XOR S5	S1 XOR S4 XOR S5 XOR S6	S2 XOR S5 XOR S6 XOR S7	S3 XOR S6 XOR S7	S4 XOR S6 XOR S7	S5 XOR S7				
Primitive					1	0	1	1	0	0	1	0	1				
XOR (S5 XOR S7)					S5 XOR S7		S5 XOR S7	S5 XOR S7			S5 XOR S7		S5 XOR S7				
		S0	S0 XOR S1	S0 XOR S1 XOR S2	S0 XOR S2 XOR S3 XOR S5 XOR S7	S1 XOR S2 XOR S3 XOR S4 XOR S6	S2 XOR S3 XOR S4 XOR S5	S0 XOR S3 XOR S4 XOR S5 XOR S6	S1 XOR S4 XOR S5 XOR S6	S2 XOR S5 XOR S6 XOR S7	S3 XOR S6 XOR S7	S4 XOR S6 XOR S7					
Primitive				1	0	1	1	0	0	1	0	1					

157

XOR (S4 XOR S6 XOR S7)					S4 XOR S6 XOR S7		S4 XOR S6 XOR S7	S4 XOR S6 XOR S7			S4 XOR S6 XOR S7		S4 XOR S6 XOR S7				
		S0	S0 XOR S1	S0 XOR S1 XOR S2	S0 XOR S2 XOR S3 XOR S4 XOR S6 XOR S7	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S7	S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S1 XOR S4 XOR S5 XOR S6 XOR S7	S2 XOR S5 XOR S6 XOR S7	S3 XOR S6 XOR S7	S4 XOR S7	S5 XOR S6					
Primitive					1	0	1	1	0	0	1	0	1				
XOR (S3 XOR S5 XOR S6					S3 XOR S5 XOR S6		S3 XOR S5 XOR S6	S3 XOR S5 XOR S6			S3 XOR S5 XOR S6		S3 XOR S5 XOR S6				
		S0	S0 XOR S1	S0 XOR S1 XOR S3	S0 XOR S1 XOR S2 XOR S3 XOR S5 XOR S6	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6	S2 XOR S3 XOR S4 XOR S5 XOR S6							

158

				S7													
Primitive		1	0	1	1	0	0	1	0	1							
XOR (S2 XOR S4 XOR S5 XOR S7)		S2 XOR S4 XOR S5 XOR S7		S2 XOR S4 XOR S5 XOR S7	S2 XOR S4 XOR S5 XOR S7			S2 XOR S4 XOR S5 XOR S7		S2 XOR S4 XOR S5 XOR S7							
		S0 XOR S2 XOR S4 XOR S5 XOR S7	S0 XOR S1 XOR S3 XOR S5 XOR S7	S0 XOR S1 XOR S2 XOR S4 XOR S6	S0 XOR S1 XOR S2 XOR S3 XOR S5 XOR S6	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S7	S2 XOR S3 XOR S4 XOR S5 XOR S6	S0 XOR S2 XOR S3 XOR S4 XOR S6 XOR S7	S1 XOR S3 XOR S4 XOR S5 XOR S6								
Primitive	1	0	1	1	0	0	1	0	1								
XOR (S1 XOR S3 XOR S4 XOR S6)	S1 XOR S3 XOR S4 XOR S6		S1 XOR S3 XOR S4 XOR S6	S1 XOR S3 XOR S4 XOR S6			S1 XOR S3 XOR S4 XOR S6		S1 XOR S3 XOR S4 XOR S6								
Final Answer (Recovered)	S1 XOR S3	S0 XOR S2	S0 XOR S4	S0 XOR S3	S0 XOR S1	S1 XOR S2	S1 XOR S2	S0 XOR S2									

	XOR S4 XOR S6	XOR S4 XOR S5	XOR S5	XOR S4 XOR S5	XOR S4 XOR S5 XOR S6	XOR S5 XOR S6 XOR S7	XOR S4 XOR S5 XOR S7	XOR S3 XOR S5									

8) S multiply with 56

Comments\bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	S0	S1	S2	S3	S4	S5	S6	S7								
x 56	0	1	1	0	1	0	1	0								
	0	0	0	0	0	0	0	0								
		S0	S1	S2	S3	S4	S5	S6	S7							
			S0	S1	S2	S3	S4	S5	S6	S7						
				0	0	0	0	0	0	0	0					
					S0	S1	S2	S3	S4	S5	S6	S7				
						0	0	0	0	0	0	0	0			
							S0	S1	S2	S3	S4	S5	S6	S7		
								0	0	0	0	0	0	0	0	
Answer		S0	S0	S1	S0	S1	S0	S1	S2	S3	S4	S5	S6	S7		
			XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR				
			S1	S2	S2	S3	S2	S3	S4	S5	S6	S7				
					XOR	XOR	XOR	XOR	XOR	XOR						
					S3	S4	S4	S5	S6	S7						
							XOR	XOR	XOR							
							S5	S6	S7							
Primitive						1	0	1	1	0	0	1	0	1		
XOR S7						S7		S7	S7			S7		S7		
		S0	S0	S1	S0	S1	S0	S1	S2	S3	S4	S5	S6			
			XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR					
			S1	S2	S2	S3	S2	S3	S4	S5	S6					
					XOR	XOR	XOR	XOR	XOR	XOR						
					S3	S4	S4	S5	S6	S7						
						XOR	XOR	XOR								
						S7	S5	S6								

								XOR								
								S7								
Primitive					1	0	1	1	0	0	1	0	1			
XOR S6					S6		S6	S6			S6		S6			
		S0	S0	S1	S0	S1	S0	S1	S2	S3	S4	S5				
			XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR						
			S1	S2	S2	S3	S2	S3	S4	S5						
					XOR	XOR	XOR	XOR	XOR	XOR						
					S3	S4	S4	S5	S6	S7						
						XOR	XOR	XOR								
						S6	S5	X7								
							S6									
Primitive				1	0	1	1	0	0	1	0	1				
XOR S5				S5		S5	S5			S5		S5				
		S0	S0	S1	S0	S1	S0	S1	S2	S3	S4					
			XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR						
			S1	S2	S2	S3	S2	S3	S4	S5						
				XOR	XOR	XOR	XOR	XOR	XOR							
				S5	S3	S4	S4	S5	S6							
					XOR	XOR	XOR	XOR								
					S6	S5	S6	X7								
						XOR										
						S7										
Primitive			1	0	1	1	0	0	1	0	1					
XOR S4			S4		S4	S4			S4		S4					
		S0	S0	S1	S0	S1	S0	S1	S2	S3						
			XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR						
			S1	S2	S2	S3	S2	S3	S6	S7						
			XOR	XOR	XOR	XOR	XOR	XOR								
			S4	S5	S3	S5	S4	S5								

					XOR	XOR	XOR	XOR								
					S4	S7	S6	X7								
					XOR											
					S6											
Primitive																
XOR S6																
Primitive																
XOR S5																
Primitive																
XOR S4																

9) S multiply with 82

Comments\bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	S0	S1	S2	S3	S4	S5	S6	S7								
x 82	0	1	0	0	0	0	0	1								
	0	0	0	0	0	0	0	0								
		S0	S1	S2	S3	S4	S5	S6	S7							
			0	0	0	0	0	0	0	0						
				0	0	0	0	0	0	0	0					
					0	0	0	0	0	0	0	0				
						0	0	0	0	0	0	0	0			
							0	0	0	0	0	0	0	0		
								S0	S1	S2	S3	S4	S5	S6	S7	
Answer		S0	S1	S2	S3	S4	S5	S0	S1	S2	S3	S4	S5	S6	S7	
								XOR	XOR							
								S6	S7							
Primitive							1	0	1	1	0	0	1	0	1	
XOR S7							S7		S7	S7			S7		S7	
		S0	S1	S2	S3	S4	S5	S0	S1	S2	S3	S4	S5	S6		
							XOR	XOR		XOR			XOR			
							S7	S6		S7			S7			
Primitive						1	0	1	1	0	0	1	0	1		
XOR S6						S6		S6	S6			S6		S6		
		S0	S1	S2	S3	S4	S5	S0	S1	S2	S3	S4	S5			
						XOR	XOR		XOR	XOR		XOR	XOR			

					XOR	XOR	XOR	XOR								
					S4	S7	S6	X7								
					XOR											
					S6											
Primitive				1	0	1	1	0	0	1	0	1				
XOR S5				S5		S5	S5			S5		S5				
		S0	S0	S1	S0	S1	S0	S1	S2	S3	S4					
			XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR						
			S1	S2	S2	S3	S2	S3	S4	S5						
				XOR	XOR	XOR	XOR	XOR	XOR							
				S5	S3	S4	S4	S5	S6							
					XOR	XOR	XOR	XOR								
					S6	S5	S6	X7								
						XOR										
						S7										
Primitive			1	0	1	1	0	0	1	0	1					
XOR S4			S4		S4	S4			S4		S4					
		S0	S0	S1	S0	S1	S0	S1	S2	S3						
			XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR						
			S1	S2	S2	S3	S2	S3	S6	S7						
			XOR	XOR	XOR	XOR	XOR	XOR								
			S4	S5	S3	S5	S4	S5								

						S6	S7		S6	S7		S6	S7				
Primitive					1	0	1	1	0	0	1	0	1				
XOR (S5 XOR S7)					S5 XOR S7		S5 XOR S7	S5 XOR S7			S5 XOR S7		S5 XOR S7				
		S0	S1	S2	S3 XOR S5 XOR S7	S4 XOR S6		S0 XOR S5 XOR S7	S1 XOR S6	S2 XOR S7	S3 XOR S5 XOR S7	S4 XOR S6					
Primitive				1	0	1	1	0	0	1	0	1					
XOR (S4 XOR S6)				S4 XOR S6		S4 XOR S6	S4 XOR S6			S4 XOR S6		S4 XOR S6					
		S0	S1	S2 XOR S4 XOR S6	S3 XOR S5 XOR S7		S4 XOR S6	S0 XOR S5 XOR S7	S1 XOR S6	S2 XOR S4 XOR S6 XOR S7	S3 XOR S5 XOR S7						
Primitive			1	0	1	1	0	0	1	0	1						
XOR (S3 XOR S5 XOR S7)			S3 XOR S5 XOR S7		S3 XOR S5 XOR S7	S3 XOR S5 XOR S7			S3 XOR S5 XOR S7		S3 XOR S5 XOR S7						
		S0	S1 XOR S3 XOR S5	S2 XOR S4		S3 XOR S5	S4 XOR S6	S0 XOR S5	S1 XOR S3	S2 XOR S4							

			S5 XOR S7	S6		S7		S7	S5 XOR S6 XOR S7	S6 XOR S7							
Primitive		1	0	1	1	0	0	1	0	1							
XOR (S2 XOR S4 XOR S6 XOR S7)		S2 XOR S4 XOR S6 XOR S7		S2 XOR S4 XOR S6 XOR S7	S2 XOR S4 XOR S6 XOR S7			S2 XOR S4 XOR S6 XOR S7		S2 XOR S4 XOR S6 XOR S7							
		S0 XOR S2 XOR S4 XOR S6 XOR S7	S1 XOR S3 XOR S5 XOR S7	S7	S2 XOR S4 XOR S6	S3 XOR S5 XOR S7	S4 XOR S6	S0 XOR S4 XOR S6	S1 XOR S3 XOR S5								
Primitive	1	0	1	1	0	0	1	0	1								
XOR (S1 XOR S3 XOR S5 XOR S6 XOR S7)	S1 XOR S3 XOR S5 XOR S6 XOR S7		S1 XOR S3 XOR S5 XOR S6 XOR S7	S1 XOR S3 XOR S5 XOR S6 XOR S7			S1 XOR S3 XOR S5 XOR S6 XOR S7		S1 XOR S3 XOR S5 XOR S6 XOR S7								

	S7		S7	S7			S7		S7								
Final Answer (Recovered)	S1 XOR S3 XOR S5 XOR S6 XOR S7	S0 XOR S2 XOR S4 XOR S6 XOR S7	S6	S1 XOR S3 XOR S5 XOR S6	S2 XOR S4 XOR S6 XOR S7	S3 XOR S5 XOR S7	S1 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S4 XOR S5 XOR S6									

10) S multiply with F3

Comments\bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	S0	S1	S2	S3	S4	S5	S6	S7								
x F3	1	1	0	0	1	1	1	1								
	S0	S1 S0	S2 S1 0	S3 S2 0	S4 S3 0 S0	S5 S4 0 S1 S0	S6 S5 0 S2 S1 S0	S7 S6 0 S3 S2 S1 S0	S7 0 S4 S3 S2 S1	0 S5 S4 S3 S2	0 S6 S5 S4 S3	S7 S6 S5 S4	S7 S6 S5	S7 S6	S7	
Answer	S0	S1 XOR S0	S1 XOR S2	S2 XOR S3	S0 XOR S3 XOR S4	S0 XOR S1 XOR S4 XOR S5	S0 XOR S1 XOR S2 XOR S5 XOR S6	S0 XOR S1 XOR S2 XOR S3 XOR S6 XOR S7	S1 XOR S2 XOR S3 XOR S4 XOR S7	S2 XOR S3 XOR S4 XOR S5	S3 XOR S4 XOR S5 XOR S6	S4 XOR S5 XOR S6 XOR S7	S5 XOR S6 XOR S7	S6 XOR S7	S7	
Primitive							1	0	1	1	0	0	1	0	1	
XOR S7							S7		S7	S7			S7		S7	
	S0	S1 XOR S0	S1 XOR S2	S2 XOR S3	S0 XOR S3	S0 XOR S1	S0 XOR S1	S0 XOR S1	S1 XOR S2	S2 XOR S3	S3 XOR S4	S4 XOR S5	S5 XOR S6	S6 XOR S7		

					XOR S4	XOR S4 XOR S5	XOR S2 XOR S3 XOR S5 XOR S6 XOR S7	XOR S2 XOR S3 XOR S6 XOR S7	XOR S3 XOR S4 XOR S5 XOR S7	XOR S4 XOR S5 XOR S6 XOR S7	XOR S5 XOR S6	XOR S6 XOR S7					
Primitive						1	0	1	1	0	0	1	0	1			
XOR (S6 XOR S7)						S6 XOR S7		S6 XOR S7	S6 XOR S7			S6 XOR S7		S6 XOR S7			
	S0	S1 XOR S0	S1 XOR S2	S2 XOR S3	S0 XOR S3 XOR S4	S0 XOR S1 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S3 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S3 XOR S4 XOR S6 XOR S7	S2 XOR S3 XOR S4 XOR S5 XOR S7	S3 XOR S4 XOR S5 XOR S6	S4 XOR S5 XOR S6	S5 XOR S6					
Primitive					1	0	1	1	0	0	1	0	1				
XOR (S6 XOR S7)					S5 XOR S6		S5 XOR S6	S5 XOR S6			S5 XOR S6		S5 XOR S6				
	S0	S1 XOR S0	S1 XOR S2	S2 XOR S3	S0 XOR S3 XOR	S0 XOR S1 XOR	S0 XOR S1 XOR	S0 XOR S1 XOR	S1 XOR S2 XOR	S2 XOR S3 XOR	S3 XOR S4	S4 XOR S5					

						S4 XOR S5 XOR S6	S4 XOR S5 XOR S6 XOR S7	S2 XOR S3 XOR S5 XOR S6 XOR S7	S2 XOR S3 XOR S4 XOR S6 XOR S7	S3 XOR S4 XOR S5 XOR S6 XOR S7	S4 XOR S5 XOR S6 XOR S7						
Primitive						1	0	1	1	0	0	1	0	1			
XOR (S4 XOR S5)						S4 XOR S5		S4 XOR S5	S4 XOR S5			S4 XOR S5		S4 XOR S5			
	S0	S1 XOR S0	S1 XOR S2	S2 XOR S3	S0 XOR S3 XOR S4 XOR S5	S0 XOR S3 XOR S4 XOR S5	S0 XOR S1 XOR S4 XOR S6	S0 XOR S1 XOR S2 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S3 XOR S4 XOR S6 XOR S7	S2 XOR S3 XOR S4 XOR S5 XOR S7	S3 XOR S4 XOR S5 XOR S6	S4 XOR S5 XOR S6	S5 XOR S6					
Primitive					1	0	1	1	0	0	1	0	1				
XOR (S3 XOR S4)					S3 XOR S4		S3 XOR S4	S3 XOR S4			S3 XOR S4		S3 XOR S4				
	S0	S1 XOR S0	S1 XOR S2	S2 XOR S3	S0 XOR S2 XOR S3	S0 XOR S1 XOR S3	S0 XOR S1 XOR S2 XOR S4	S0 XOR S1 XOR S2 XOR S3 XOR S6	S1 XOR S2 XOR S3 XOR S4 XOR S6	S2 XOR S3 XOR S4 XOR S5 XOR S6	S3 XOR S4 XOR S5 XOR S6	S4 XOR S5 XOR S6					

			XOR S4	XOR S5		XOR S4 XOR S6 XOR S7	XOR S4 XOR S5 XOR S6 XOR S7	XOR S3 XOR S5 XOR S6 XOR S7	XOR S3 XOR S7								
Primitive		1	0	1	1	0	0	1	0	1							
		S2 XOR S3 XOR S7		S2 XOR S3 XOR S7	S2 XOR S3 XOR S7			S2 XOR S3 XOR S7		S2 XOR S3 XOR S7							
	S0	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S7	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S4 XOR S5 XOR S7	S0 XOR S2 XOR S3 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S3 XOR S4 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S3 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S3 XOR S4 XOR S6 XOR S7								
Primitive	1	0	1	1	0	0	1	0	1								
XOR (S1 XOR S2 XOR S6 XOR S7)	S1 XOR S2 XOR S6 XOR S7		S1 XOR S2 XOR S6 XOR S7	S1 XOR S2 XOR S6 XOR S7			S1 XOR S2 XOR S6 XOR S7		S1 XOR S2 XOR S6 XOR S7								

Final Answer (Recovered)	S0 XOR S1 XOR S2 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S3 XOR S7	S3 XOR S4 XOR S6 XOR S7	S1 XOR S2 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S3 XOR S5 XOR S6 XOR S7									

11) S multiply with 1E

Comments\bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	S0	S1	S2	S3	S4	S5	S6	S7								
x 1E	0	1	1	1	1	0	0	0								
	0	0	0	0	0	0	0	0								
		S0	S1	S2	S3	S4	S5	S6	S7							
			S0	S1	S2	S3	S4	S5	S6	S7						
				S0	S1	S2	S3	S4	S5	S6	S7					
					0	0	0	0	0	0	0	0	0			
						0	0	0	0	0	0	0	0	0		
								0	0	0	0	0	0	0	0	
Answer		S0	S0	S0	S0	S1	S2	S3	S4	S5	S6	S7				
			XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR					
			S1	S1	S1	S2	S3	S4	S5	S6	S7					
				XOR	XOR	XOR	XOR	XOR	XOR	XOR						
				S2	S2	S3	S4	S5	S6	S7						
					XOR	XOR	XOR	XOR	XOR							
					S3	S4	S5	S6	S7							
Primitive				1	0	1	1	0	0	1	0	1				
XOR S7				S7		S7	S7			S7		S7				
		S0	S0	S0	S0	S1	S2	S3	S4	S5	S6					
			XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR					
			S1	S1	S1	S2	S3	S4	S5	S6	S7					
				XOR	XOR	XOR	XOR	XOR	XOR	XOR						
				S2	S2	S3	S4	S5	S6							
				XOR	XOR	XOR	XOR	XOR	XOR							
				S7	S3	S4	S5	S6	S7							

				S7	S7											
Primitive	1	0	1	1	0	0	1	0	1							
XOR (S4 XOR S5)	S4		S4	S4			S4		S4							
	XOR		XOR	XOR			XOR		XOR							
	S5		S5	S5			S5		S5							
Final Answer	S4	S0	S0	S0	S0	S1	S2	S3								
(Recovered)	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR								
	S5	S5	S1	S1	S1	S2	S3	S4								
		XOR	XOR	XOR	XOR	XOR	XOR									
		S6	S4	S2	S2	S3	S7									
			XOR	XOR	XOR	XOR										
			S5	S4	S3	S4										
			XOR	XOR	XOR	XOR										
			S6	S6	S5	S6										
			XOR	XOR	XOR											
			S7	S7	S7											

						XOR	XOR									
						S7	S7									
Primitive			1	0	1	1	0	0	1	0	1					
XOR (S6 XOR S7)			S6		S6	S6			S6		S6					
			XOR		XOR	XOR			XOR		XOR					
			S7		S7	S7			S7		S7					
		S0	S0	S0	S0	S1	S2	S3	S4	S5						
			XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR						
			S1	S1	S1	S2	S3	S4	S5	S6						
			XOR	XOR	XOR	XOR	XOR	XOR								
			S6	S2	S2	S3	S4	S5								
			XOR	XOR	XOR	XOR	XOR	XOR								
			S7	S7	S3	S4	S5	S6								
					XOR	XOR	XOR									
					S6	S6	S7									
					XOR											
					S7											
Primitive		1	0	1	1	0	0	1	0	1						
		S5		S5	S5			S5		S5						
		XOR		XOR	XOR			XOR		XOR						
		S6		S6	S6			S6		S6						
		S0	S0	S0	S0	S1	S2	S3	S4							
		XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR							
		S5	S1	S1	S1	S2	S3	S4	S5							
		XOR	XOR	XOR	XOR	XOR	XOR									
		S6	S6	S2	S2	S3	S4									
			XOR	XOR	XOR	XOR	XOR									
			S7	S5	S3	S4	S5									
				XOR	XOR	XOR	XOR									
				S6	S5	S6	S7									
				XOR	XOR											

12) S multiply with C6

Comments\bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	S0	S1	S2	S3	S4	S5	S6	S7								
x C6	0	1	1	0	0	0	1	1								
	0	0	0	0	0	0	0	0								
		S0	S1	S2	S3	S4	S5	S6	S7							
			S0	S1	S2	S3	S4	S5	S6	S7						
				0	0	0	0	0	0	0	0					
					0	0	0	0	0	0	0	0				
						0	0	0	0	0	0	0	0			
							S0	S1	S2	S3	S4	S5	S6	S7		
								S0	S1	S2	S3	S4	S5	S6	S7	
Answer		S0	S0	S1	S2	S3	S0	S0	S1	S2	S3	S4	S5	S6	S7	
			XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR		
			S1	S2	S3	S4	S4	S1	S2	S3	S4	S5	S5	S7		
							XOR	XOR	XOR	XOR						
							S5	S5	S6	S7						
								XOR	XOR							
								S6	S7							
Primitive							1	0	1	1	0	0	1	0	1	
XOR S7							S7		S7	S7			S7		S7	
		S0	S0	S1	S2	S3	S0	S0	S1	S2	S3	S4	S5	S6		
			XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR		
			S1	S2	S3	S4	S4	S1	S2	S3	S4	S5	S6	S7		
							XOR	XOR	XOR				XOR			
							S5	S5	S6				S7			
								XOR	XOR							
								S7	S6							

13) S multiply with 68

Comments\bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	S0	S1	S2	S3	S4	S5	S6	S7								
X 68	0	0	0	1	0	1	1	0								
	0	0	0	0	0	0	0	0								
		0	0	0	0	0	0	0	0							
			0	0	0	0	0	0	0	0						
				S0	S1	S2	S3	S4	S5	S6	S7					
					0	0	0	0	0	0	0	0				
						S0	S1	S2	S3	S4	S5	S6	S7			
							S0	S1	S2	S3	S4	S5	S6	S7		
								0	0	0	0	0	0	0	0	
Answer				S0	S1	S0	S0	S1	S2	S3	S4	S5	S6	S7		
						XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR			
						S2	S1	S2	S3	S4	S5	S6	S7			
							XOR	XOR	XOR	XOR	XOR					
							S3	S4	S5	S6	S7					
Primitive						1	0	1	1	0	0	1	0	1		
XOR S7						S7		S7	S7			S7		S7		
				S0	S1	S0	S0	S1	S2	S3	S4	S5	S6			
						XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR			
						S2	S1	S2	S3	S4	S5	S6	S7			
						XOR	XOR	XOR	XOR	XOR	XOR	XOR				
						S7	S3	S4	S5	S6	S7	S7				
								XOR	XOR	XOR						
								S5	S5	S6						
								XOR	XOR							
								S6	S7							
Primitive					1	0	1	1	0	0	1	0	1			
XOR (S6 XOR S7)					S6		S6	S6			S6		S6			

									XOR		XOR	XOR			XOR		XOR
									S7		S7	S7			S7		S7
						S0		S1	S0	S0	S0	S1	S2	S3	S4	S5	
								XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	
								S6	S2	S2	S1	S2	S3	S4	S5	S6	
								XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	
								S7	S7	S7	S3	S4	S5	S6	S6	S7	
											XOR	XOR	XOR				
											S6	S6	S7				
											XOR						
											S7						
Primitive						1	0	1	1	0	0	1	0	1			
XOR (S5 XOR S6 XOR S7)						S5		S5	S5			S5		S5			
						XOR		XOR	XOR			XOR		XOR			
						S6		S6	S6			S6		S6			
						XOR		XOR	XOR			XOR		XOR			
						S7		S7	S7			S7		S7			
						S0	S1	S0	S0	S1	S2	S3	S4				
						XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR				
						S5	S6	S2	S1	S2	S3	S4	S5				
						XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR				
						S6	S7	S5	S3	S4	S5	S5	S6				
						XOR		XOR	XOR	XOR	XOR	XOR					
						S7		S6	S5	S6	S7	S7					
Primitive			1			0	1	1	0	0	1	0	1				
XOR (S4 XOR S5 S6)			S4				S4	S4				S4		S4			
			XOR				XOR	XOR				XOR		XOR			
			S5				S5	S5				S5		S5			
			XOR				XOR	XOR				XOR		XOR			
			S6				S6	S6				S6		S6			
			S4	S0	S1	S0	S0	S1	S2	S3							

			XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR						
			S5	S5	S4	S2	S1	S2	S3	S4						
			XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR						
			S6	S6	S5	S3	S2	S3	S4	S5						
				XOR	XOR	XOR	XOR	XOR	XOR	XOR						
				S7	S7	S4	S3	S4	S5	S6						
										S7						
Primitive																
XOR (S3 XOR S4 XOR S5 XOR S7)			S3		S3	S3			S3		S3					
			XOR		XOR	XOR			XOR		XOR					
			S4		S4	S4			S4		S4					
			XOR		XOR	XOR			XOR		XOR					
			S5		S5	S5			S5		S5					
			XOR		XOR	XOR			XOR		XOR					
			S7		S7	S7			S7		S7					
		S3	S4	S0	S1	S0	S0	S1	S2							
		XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR							
		S4	S5	S3	S3	S2	S1	S2	S3							
		XOR	XOR	XOR		XOR	XOR	XOR	XOR							
		S5	S6	S4		S4	S3	S3	S4							
		XOR		XOR			XOR	XOR	XOR							
		S7		S6			S5	S5	S6							
								XOR	XOR							
								S6	S7							
								XOR								
								S7								
Primitive	1	0	1	1	0	0	1	0	1							
XOR (S2 XOR S3 XOR S4 XOR S6 XOR S7)	S2		S2	S2			S2		S2							
	XOR		XOR	XOR			XOR		XOR							
	S3		S3	S3			S3		S3							

	XOR		XOR	XOR			XOR		XOR							
	S4		S4	S4			S4		S4							
	XOR		XOR	XOR			XOR		XOR							
	S6		S6	S6			S6		S6							
	XOR		XOR	XOR			XOR		XOR							
	S7		S7	S7			S7		S7							
Final Answer (recovered)	S2	S3	S2	S0	S1	S0	S0	S1								
	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR								
	S3	S4	S3	S2	S3	S2	S1	S2								
	XOR	XOR	XOR	XOR		XOR	XOR	XOR								
	S4	S5	S5	S7		S4	S2	S3								
	XOR	XOR	XOR				XOR	XOR								
	S6	S7	S7				S4	S5								
	XOR						XOR	XOR								
	S7						S5	S6								
							XOR	XOR								
							S6	S7								
							XOR									
							S7									

14) S multiply with E5

Comments\bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	S0	S1	S2	S3	S4	S5	S6	S7								
x E5	1	0	1	0	0	1	1	1								
	S0	S1	S2	S3	S4	S5	S6	S7								
		0	0	0	0	0	0	0	0							
			S0	S1	S2	S3	S4	S5	S6	S7						
				0	0	0	0	0	0	0	0	0				
					0	0	0	0	0	0	0	0				
						S0	S1	S2	S3	S4	S5	S6	S7			
							S0	S1	S2	S3	S4	S5	S6	S7		
								S0	S1	S2	S3	S4	S5	S6	S7	
Answer	S0	S1	S0	S1	S2	S0	S0	S0	S1	S2	S3	S4	S5	S6	S7	
			XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR		
			S2	S3	S4	S3	S1	S1	S2	S3	S4	S5	S6	S7		
						XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR			
						S5	S4	S2	S3	S4	S5	S6	S7			
							XOR	XOR	XOR	XOR						
							S6	S5	S6	S7						
								XOR								
								S7								
Primitive							1	0	1	1	0	0	1	0	1	
XOR S7							S7		S7	S7			S7		S7	
	S0	S1	S0	S1	S2	S0	S0	S0	S1	S2	S3	S4	S5	S6		
			XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR		
			S2	S3	S4	S3	S1	S1	S2	S3	S4	S5	S6	S7		
						XOR	XOR	XOR	XOR	XOR	XOR	XOR				
						S5	S4	S2	S3	S4	S5	S6				

								XOR	XOR	XOR						
								S6	S5	S6						
								XOR	XOR	XOR						
								S7	S7	S7						
Primitive							1	0	1	1	0	0	1	0	1	
XOR (S6 XOR S7)							S6		S6	S6			S6		S6	
							XOR		XOR	XOR			XOR		XOR	
							S7		S7	S7			S7		S7	
	S0	S1	S0	S1	S2	S0	S0	S0	S1	S2	S3	S4	S5			
			XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR			
			S2	S3	S4	S3	S1	S1	S2	S3	S4	S5	S6			
						XOR	XOR	XOR	XOR	XOR	XOR	XOR				
						S5	S4	S2	S3	S4	S5	S7				
							XOR	XOR	XOR	XOR						
							S6	S5	S6	S7						
								XOR								
								S7								
Primitive							1	0	1	1	0	0	1	0	1	
XOR (S5 XOR S6)							S5		S5	S5			S5		S5	
							XOR		XOR	XOR			XOR		XOR	
							S6		S6	S6			S6		S6	
	S0	S1	S0	S1	S2	S0	S0	S0	S1	S2	S3	S4				
			XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR				
			S2	S3	S4	S3	S1	S1	S2	S3	S4	S5				
						XOR	XOR	XOR	XOR	XOR	XOR	XOR				
						S5	S4	S2	S3	S4	S6	S7				
							XOR	XOR								
							S6	S5								
								XOR								
								S7								
Primitive						1	0	1	1	0	0	1	0	1		

XOR (S4 XOR S5 XOR S7)				S4		S4	S4			S4		S4				
				XOR		XOR	XOR			XOR		XOR				
				S5		S5	S5			S5		S5				
				XOR		XOR	XOR			XOR		XOR				
				S7		S7	S7			S7		S7				
	S0	S1	S0	S1	S2	S0	S0	S0	S1	S2	S3					
			XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR					
			S2	S3	S4	S3	S1	S1	S2	S3	S4					
				XOR	XOR	XOR		XOR	XOR	XOR	XOR					
				S4	S5	S4		S2	S3	S5	S5					
				XOR	XOR	XOR				XOR	XOR					
				S5	S6	S5				S7	S6					
				XOR		XOR					XOR					
				S7		S7					S7					
Primitive			1	0	1	1	0	0	1	0	1					
XOR (S3 XOR S4 XOR S6)			S3		S3	S3			S3		S3					
			XOR		XOR	XOR			XOR		XOR					
			S4		S4	S4			S4		S4					
			XOR		XOR	XOR			XOR		XOR					
			S6		S6	S6			S6		S6					
	S0	S1	S0	S1	S2	S0	S0	S0	S1	S2						
			XOR	XOR	XOR		XOR	XOR	XOR	XOR						
			S2	S3	S3		S1	S1	S2	S3						
			XOR	XOR	XOR			XOR	XOR	XOR						
			S3	S4	S5			S2	S4	S5						
			XOR	XOR					XOR	XOR						
			S4	S5					S6	S7						
			XOR	XOR												
			S6	S7												
Primitive		1	0	1	1	0	0	1	0	1						
XOR (S2 XOR S3		S2		S2	S2			S2		S2						
		S3		S3	S3			S3		S3						

		XOR		XOR	XOR			XOR		XOR						
		S5		S5	S5			S5		S5						
		XOR		XOR	XOR			XOR		XOR						
		S7		S7	S7			S7		S7						
	S0	S1	S0	S1	S7	S0	S0	S0	S1							
		XOR	XOR	XOR			XOR	XOR	XOR							
		S2	S2	S2			S1	S1	S2							
		XOR	XOR	XOR					XOR							
		S3	S3	S4					S3							
		XOR	XOR						XOR							
		S5	S4						S5							
		XOR	XOR						XOR							
		S7	S6						S7							
Primitive	1	0	1	1	0	0	1	0	1							
XOR (S1 XOR S2 XOR S4 XOR S6)	S1		S1	S1			S1		S1							
	XOR		XOR	XOR			XOR		XOR							
	S2		S2	S2			S2		S2							
	XOR		XOR	XOR			XOR		XOR							
	S4		S4	S4			S4		S4							
	XOR		XOR	XOR			XOR		XOR							
	S6		S6	S6			S6		S6							
Final Answer (Recovered)	S0	S1	S0	S6	S7	S0	S0	S0								
	XOR	XOR	XOR				XOR	XOR								
	S1	S2	S1				S2	S1								
	XOR	XOR	XOR				XOR	XOR								
	S2	S3	S3				S4	S3								
	XOR	XOR					XOR	XOR								
	S4	S5					S6	S5								
	XOR	XOR						XOR								
	S6	S7						S7								

15) S multiply with 02

Comments\bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	S0	S1	S2	S3	S4	S5	S6	S7								
x 02	0	1	0	0	0	0	0	0								
	0	0	0	0	0	0	0	0								
		S0	S1	S2	S3	S4	S5	S6	S7							
			0	0	0	0	0	0	0	0						
				0	0	0	0	0	0	0	0					
					0	0	0	0	0	0	0	0				
						0	0	0	0	0	0	0	0			
							0	0	0	0	0	0	0	0		
								0	0	0	0	0	0	0	0	
Answer		S0	S1	S2	S3	S4	S5	S6	S7							
Primitive	1	0	1	1	0	0	1	0	1							
XOR S7	S7		S7	S7			S7		S7							
Final Answer (Recovered)	S7	S0	S1 XOR S7	S2 XOR S7	S3	S4	S5 XOR S7	S6								

16) S multiply with A1

Comments\bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	S0	S1	S2	S3	S4	S5	S6	S7								
x A1	1	0	0	0	0	1	0	1								
	S0	S1	S2	S3	S4	S5	S6	S7								
		0	0	0	0	0	0	0	0							
			0	0	0	0	0	0	0	0						
				0	0	0	0	0	0	0	0					
					0	0	0	0	0	0	0	0				
						0	0	0	0	0	0	0	0			
							S0	S1	S2	S3	S4	S5	S6	S7		
								0	0	0	0	0	0	0	0	
									S0	S1	S2	S3	S4	S5	S6	S7
Answer	S0	S1	S2	S3	S4	S0 XOR S5	S1 XOR S6	S0 XOR S2 XOR S7	S1 XOR S3	S2 XOR S4	S3 XOR S5	S4 XOR S6	S5 XOR S7	S6	S7	
Primitive							1	0	1	1	0	0	1	0	1	
XOR S7							S7		S7	S7			S7		S7	
	S0	S1	S2	S3	S4	S0 XOR S5	S1 XOR S6	S0 XOR S2 XOR S7	S1 XOR S3	S2 XOR S4	S3 XOR S5	S4 XOR S6	S5	S6		
Primitive						1	0	1	1	0	0	1	0	1		
XOR S6						S6		S6	S6			S6		S6		
	S0	S1	S2	S3	S4	S0	S1	S0	S1	S2	S3	S4	S5			

						XOR S5 XOR S6	XOR S6 XOR S7	XOR S2 XOR S6 XOR S7	XOR S3 XOR S6 XOR S7	XOR S4 XOR S7	XOR S5					
Primitive					1	0	1	1	0	0	1	0	1			
XOR S5					S5		S5	S5			S5		S5			
	S0	S1	S2	S3	S4 XOR S5	S0 XOR S5 XOR S6	S1 XOR S5 XOR S6 XOR S7	S0 XOR S2 XOR S6 XOR S7	S1 XOR S3 XOR S6 XOR S7	S2 XOR S4 XOR S7	S3	S4				
Primitive				1	0	1	1	0	0	1	0	1				
XOR S4				S4		S4	S4			S4		S4				
	S0	S1	S2	S3 XOR S4	S4 XOR S5	S0 XOR S4 XOR S5 XOR S6	S1 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S2 XOR S5 XOR S6 XOR S7	S1 XOR S3 XOR S6 XOR S7	S2 XOR S7	S3					
Primitive			1	0	1	1	0	0	1	0	1					
XOR (S3)			S3		S3	S3			S3		S3					

	S0	S1	S2 XOR S3	S3 XOR S4	S3 XOR S4 XOR S5	S0 XOR S3 XOR S4 XOR S5 XOR S6	S1 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S2 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S6 XOR S7	S2 XOR S7						
Primitive		1	0	1	1	0	0	1	0	1						
XOR (S2 XOR S7)		S2 XOR S7		S2 XOR S7	S2 XOR S7			S2 XOR S7		S2 XOR S7						
	S0	S1 XOR S2 XOR S7	S2 XOR S3	S2 XOR S3 XOR S4	S2 XOR S3 XOR S4 XOR S5	S0 XOR S3 XOR S4 XOR S5 XOR S6	S1 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S5 XOR S6 XOR S7	S1 XOR S6 XOR S7							
Primitive	1	0	1	1	0	0	1	0	1							
XOR (S1 XOR S6 XOR S7)	S1 XOR S6		S1 XOR S6	S1 XOR S6			S1 XOR S6		S1 XOR S6							

	XOR S7		XOR S7	XOR S7			XOR S7		XOR S7							
Final Answer (Recovered)	S0 XOR S1 XOR S6 XOR S7	S1 XOR S2 XOR S7	S1 XOR S2 XOR S3 XOR S6 XOR S7	S1 XOR S2 XOR S3 XOR S4 XOR S6 XOR S7	S2 XOR S3 XOR S4 XOR S5 XOR S7	S0 XOR S3 XOR S4 XOR S5 XOR S6	S4 XOR S5	S0 XOR S5 XOR S6								

17) S multiply with FC

Comments\bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	S0	S1	S2	S3	S4	S5	S6	S7								
x FC	0	0	1	1	1	1	1	1								
	0	0	0	0	0	0	0	0								
		0	0	0	0	0	0	0	0							
			S0	S1	S2	S3	S4	S5	S6	S7						
				S0	S1	S2	S3	S4	S5	S6	S7					
					S0	S1	S2	S3	S4	S5	S6	S7				
						S0	S1	S2	S3	S4	S5	S6	S7			
							S0	S1	S2	S3	S4	S5	S6	S7		
								S0	S1	S2	S3	S4	S5	S6	S7	
Answer			S0	S0 XOR S1	S0 XOR S1 XOR S2	S0 XOR S1 XOR S2 XOR S3	S0 XOR S1 XOR S2 XOR S3 XOR S4	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S3 XOR S4 XOR S5 XOR S6 XOR S7	S4 XOR S5 XOR S6 XOR S7	S5 XOR S6 XOR S7	S6 XOR S7	S7	
Primitive							1	0	1	1	0	0	1	0	1	
XOR S7							S7		S7	S7			S7		S7	
			S0	S0 XOR S1	S0 XOR S1 XOR S2	S0 XOR S1 XOR S2 XOR S3	S0 XOR S1 XOR S2 XOR S3 XOR S4	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6	S2 XOR S3 XOR S4 XOR S5 XOR S6	S3 XOR S4 XOR S5 XOR S6	S4 XOR S5 XOR S6	S5 XOR S6	S6	S7	

					XOR S2	XOR S2 XOR S3	XOR S2 XOR S3 XOR S4 XOR S7	XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	XOR S4 XOR S5 XOR S6 XOR S7	XOR S5 XOR S6 XOR S7	XOR S6 XOR S7				
Primitive						1	0	1	1	0	0	1	0	1		
XOR (S6 XOR S7)						S6 XOR S7		S6 XOR S7	S6 XOR S7			S6 XOR S7		S6 XOR S7		
			S0	S0 XOR S1	S0 XOR S1 XOR S2	S0 XOR S1 XOR S2 XOR S3 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S3 XOR S4 XOR S5 XOR S6 XOR S7	S4 XOR S5 XOR S6 XOR S7	S5 XOR S6			
Primitive					1	0	1	1	0	0	1	0	1			
XOR (S5 XOR S6)					S5		S5	S5			S5		S5			

						XOR S6		XOR S5	XOR S6			XOR S6		XOR S6		
			S0	S0 XOR S1	S0 XOR S1 XOR S2	S0 XOR S1 XOR S2 XOR S5 XOR S6	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S3 XOR S4 XOR S5 XOR S6 XOR S7	S4 XOR S5 XOR S6 XOR S7	S5 XOR S6			
Primitive				1	0	1	1	0	0	1	0	1				
XOR (S4 XOR S5)				S4 XOR S5		S4 XOR S5	S4 XOR S5			S4 XOR S5		S4 XOR S5				
			S0	S0 XOR S1	S0 XOR S1 XOR S4 XOR S5	S0 XOR S1 XOR S2 XOR S3 XOR S6	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S3 XOR S4 XOR S5 XOR S6 XOR S7						

	S0	S1	S2	S3	S4	S5 XOR S6 XOR S7	S0 XOR S6 XOR S7	S0 XOR S1 XOR S6	S1 XOR S2 XOR S6	S2 XOR S3 XOR S7	S3 XOR S4	S4 XOR S5 XOR S6 XOR S7	S5 XOR S6 XOR S7				
Primitive					1	0	1	1	0	0	1	0	1				
XOR (S5 XOR S6 XOR S7)					S5 XOR S6 XOR S7		S5 XOR S6 XOR S7	S5 XOR S6 XOR S7			S5 XOR S6 XOR S7		S5 XOR S6 XOR S7				
	S0	S1	S2	S3	S4 XOR S5 XOR S6 XOR S7	S5 XOR S6 XOR S7	S0 XOR S5	S0 XOR S1 XOR S5 XOR S7	S1 XOR S2 XOR S6	S2 XOR S3 XOR S7	S3 XOR S4 XOR S5 XOR S6 XOR S7	S4 XOR S5 XOR S6 XOR S7					
Primitive				1	0	1	1	0	0	1	0	1					
XOR (S4 XOR S5 XOR S6 XOR S7)				S4 XOR S5 XOR S6 XOR S7		S4 XOR S5 XOR S6 XOR S7	S4 XOR S5 XOR S6 XOR S7			S4 XOR S5 XOR S6 XOR S7		S4 XOR S5 XOR S6 XOR S7					
	S0	S1	S2	S3	S4	S4	S0	S0	S1	S2	S3						

201

				XOR S4 XOR S5 XOR S6 XOR S7	XOR S5 XOR S6 XOR S7		XOR S4 XOR S5 XOR S6 XOR S7	XOR S1 XOR S5 XOR S6 XOR S7	XOR S2 XOR S5 XOR S6 XOR S7	XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	XOR S4 XOR S5 XOR S6 XOR S7						
Primitive				1	0	1	1	0	0	1	0	1					
XOR (S3 XOR S4 XOR S5 XOR S6 XOR S7)			S3 XOR S4 XOR S5 XOR S6 XOR S7		S3 XOR S4 XOR S5 XOR S6 XOR S7	S3 XOR S4 XOR S5 XOR S6 XOR S7			S3 XOR S4 XOR S5 XOR S6 XOR S7		S3 XOR S4 XOR S5 XOR S6 XOR S7						
	S0	S1	S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S3 XOR S4 XOR S5 XOR S6 XOR S7	S3	S3 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S6 XOR S7	S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S3 XOR S4 XOR S5 XOR S6 XOR S7						
Primitive		1	0	1	1	0	0	1	0	1	0	1					
XOR (S2 XOR S3		S2		S2	S2			S2		S2							

202

XOR S4 XOR S5 XOR S6)		XOR S3 XOR S4 XOR S5 XOR S6		XOR S3 XOR S4 XOR S5 XOR S6	XOR S3 XOR S4 XOR S5 XOR S6			XOR S3 XOR S4 XOR S5 XOR S6		XOR S3 XOR S4 XOR S5 XOR S6							
	S0	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6	S2 XOR S3 XOR S4 XOR S5 XOR S6	S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S3 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S6 XOR S7								
Primitive	1	0	1	1	0	0	1	0	1								
XOR (S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S7)	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR		S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR			S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR		S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR								

	S5 XOR S7		S5 XOR S7	S5 XOR S7			S5 XOR S7		S5 XOR S7								
Final Answer (Recovered)	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6	S1 XOR S2 XOR S6	S1 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S3 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S3 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S6 XOR S7								

19) S multiply with 47

Comments\bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	S0	S1	S2	S3	S4	S5	S6	S7								
x 47	1	1	1	0	0	0	1	0								
	S0	S1 S0	S2 S1 S0	S3 S2 S1 0	S4 S3 S2 0	S5 S4 S3 0	S6 S5 S4 0	S7 S6 S5 0	S7 S6 0	S7 0	0	0	0			
						0	0	0	0	0	0	0	0			
						0	0	0	0	0	0	0	0			
							S0	S1	S2	S3	S4	S5	S6	S7		
Answer	S0	S0 XOR S1	S0 XOR S1 XOR S2	S1 XOR S2 XOR S3	S2 XOR S3 XOR S4	S3 XOR S4 XOR S5	S0 XOR S4 XOR S5 XOR S6	S1 XOR S5 XOR S6 XOR S7	S2 XOR S6 XOR S7	S3 XOR S7	S4	S5	S6	S7	0	0
Primitive						1	0	1	1	0	0	1	0	1		
XOR S7						S7		S7	S7			S7		S7		
	S0	S0 XOR S1	S0 XOR S1 XOR S2	S1 XOR S2 XOR S3	S2 XOR S3 XOR S4	S3 XOR S4 XOR S5 XOR S7	S0 XOR S4 XOR S5 XOR S6	S1 XOR S5 XOR S6 XOR S7	S2 XOR S6 XOR S7	S3 XOR S7	S4	S5 XOR S7	S6			

Primitive					1	0	1	1	0	0	1	0	1			
XOR S6					S6		S6	S6			S6		S6			
	S0	S0 XOR S1	S0 XOR S1 XOR S2	S1 XOR S2 XOR S3	S2 XOR S3 XOR S4 XOR S6	S3 XOR S4 XOR S5 XOR S7	S0 XOR S4 XOR S5	S1 XOR S5 XOR S6	S2 XOR S6 XOR S7	S3 XOR S7	S4 XOR S6	S5 XOR S7				
Primitive				1	0	1	1	0	0	1	0	1				
XOR (S5 XOR S7)				S5 XOR S7		S5 XOR S7	S5 XOR S7			S5 XOR S7		S5 XOR S7				
	S0	S0 XOR S1	S0 XOR S1 XOR S2	S1 XOR S2 XOR S3 XOR S5 XOR S7	S2 XOR S3 XOR S4 XOR S6	S3 XOR S4	S0 XOR S4 XOR S7	S1 XOR S5	S2 XOR S6	S3 XOR S5	S4 XOR S6					
Primitive			1	0	1	1	0	0	1	0	1					
XOR (S4 XOR S6)			S4 XOR S6		S4 XOR S6	S4 XOR S6			S4 XOR S6		S4 XOR S6					
	S0	S0 XOR S1	S0 XOR S1 XOR S2	S1 XOR S2 XOR S3	S2 XOR S3	S3 XOR S6	S0 XOR S4 XOR S7	S1 XOR S5	S2 XOR S4	S3 XOR S5						

			XOR S4 XOR S6	XOR S5 XOR S7												
Primitive		1	0	1	1	0	0	1	0	1						
XOR (S3 XOR S5)		S3 XOR S5		S3 XOR S5	S3 XOR S5			S3 XOR S5		S3 XOR S5						
	S0	S0 XOR S1 XOR S3 XOR S5	S0 XOR S1 XOR S2 XOR S4 XOR S6	S1 XOR S2 XOR S7	S2 XOR S5	S3 XOR S6	S0 XOR S4 XOR S7	S1 XOR S3	S2 XOR S4							
Primitive	1	0	1	1	0	0	1	0	1							
XOR (S2 XOR S4)	S2 XOR S4		S2 XOR S4	S2 XOR S4			S2 XOR S4		S2 XOR S4							
Final Answer (Recovered)	S0 XOR S2 XOR S4	S0 XOR S1 XOR S3 XOR S5	S0 XOR S1 XOR S6	S1 XOR S4 XOR S7	S2 XOR S5	S3 XOR S6	S0 XOR S2 XOR S7	S1 XOR S3								

20) S multiply with AE

Comments\bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	S0	S1	S2	S3	S4	S5	S6	S7								
x AE	0	1	1	1	0	1	0	1								
	0	0 S0	0 S1 S0	0 S2 S1 S0	0 S3 S2 S1 0	0 S4 S3 S2 0 S0	0 S5 S4 S3 0 S1	0 S6 S5 S4 0 S2 0	0 S7 S6 S5 0 S3 0 S0	S7 S6 S5 0 S4 0 S1	S7 S6 0 S5 0 S2	S7 0 S6 0 S3	0 S7 0 S4	0 S5	S6	S7
Answer		S0	S0 XOR S1	S0 XOR S1 XOR S2	S1 XOR S2 XOR S3 XOR S4	S0 XOR S3 XOR S4 XOR S5	S1 XOR S3 XOR S4 XOR S5 XOR S6	S0 XOR S2 XOR S4 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S3 XOR S4 XOR S5 XOR S6 XOR S7	S4 XOR S5 XOR S6 XOR S7	S5 XOR S6 XOR S7	S6 XOR S7	S7
Primitive							1	0	1	1	0	0	1	0	1	
XOR S7							S7		S7	S7			S7		S7	
		S0	S0 XOR S1	S0 XOR S1 XOR	S1 XOR S2 XOR	S0 XOR S2 XOR	S1 XOR S3 XOR	S0 XOR S2 XOR	S1 XOR S3 XOR	S2 XOR S4 XOR	S3 XOR S5 XOR	S4 XOR S6	S5	S6		

				S2	S3	S3 XOR S4	S4 XOR S5 XOR S7	S4 XOR S5 XOR S6	S5 XOR S6	S6	S7						
Primitive						1	0	1	1	0	0	1	0	1			
XOR S6						S6		S6	S6			S6		S6			
		S0	S0 XOR S1	S0 XOR S1 XOR S2	S1 XOR S2 XOR S3	S0 XOR S2 XOR S3 XOR S4 XOR S6 XOR S7	S1 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S2 XOR S4 XOR S5 XOR S6	S1 XOR S3 XOR S5 XOR S6	S2 XOR S4 XOR S6	S3 XOR S5 XOR S7	S4	S5				
Primitive					1	0	1	1	0	0	1	0	1				
XOR S5					S5		S5	S5			S5		S5				
		S0	S0 XOR S1	S0 XOR S1 XOR S2	S1 XOR S2 XOR S3	S0 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6	S1 XOR S3 XOR S4 XOR S5 XOR S7	S0 XOR S2 XOR S4 XOR S5 XOR S6	S1 XOR S3 XOR S5 XOR S6	S2 XOR S4 XOR S6	S3 XOR S5 XOR S7	S4					
Primitive				1	0	1	1	0	0	1	0	1					
XOR S4				S4		S4	S4			S4		S4					
		S0	S0 XOR	S0 XOR	S1 XOR	S0 XOR	S1 XOR	S0 XOR	S1 XOR	S2 XOR	S3 XOR						

			S1	S1 XOR S2 XOR S4	S2 XOR S3 XOR S5	S2 XOR S3 XOR S5	S3 XOR S7	S2 XOR S4	S3 XOR S5	S6	S7						
Primitive			1	0	1	1	0	0	1	0	1						
XOR (S3 XOR S7)			S3 XOR S7		S3 XOR S7	S3 XOR S7			S3 XOR S7		S3 XOR S7						
		S0	S0 XOR S1 XOR S3 XOR S7	S0 XOR S1 XOR S2 XOR S4	S1 XOR S2 XOR S5 XOR S6 XOR S7	S0 XOR S2 XOR S5 XOR S6 XOR S7	S1 XOR S3 XOR S7	S0 XOR S2 XOR S4	S1 XOR S3 XOR S5 XOR S7	S2 XOR S6							
Primitive		1	0	1	1	0	0	1	0	1							
XOR (S2 XOR S6)		S2 XOR S6		S2 XOR S6	S2 XOR S6			S2 XOR S6		S2 XOR S6							
		S0 XOR S2 XOR S6	S0 XOR S1 XOR S3 XOR S7	S0 XOR S1 XOR S4 XOR S6 XOR S7	S1 XOR S5 XOR S6 XOR S7	S0 XOR S2 XOR S6 XOR S7	S1 XOR S3 XOR S7	S0 XOR S4 XOR S6	S1 XOR S5 XOR S7								
Primitive	1	0	1	1	0	0	1	0	1								
XOR (S1 XOR S5	S1		S1	S1			S1		S1								

XOR S7)	XOR S5 XOR S7		XOR S5 XOR S7	XOR S5 XOR S7			XOR S5 XOR S7		XOR S5 XOR S7								
Final Answer (Recovered)	S1 XOR S5 XOR S7	S0 XOR S2 XOR S6	S0 XOR S3 XOR S5	S0 XOR S4 XOR S5 XOR S6 XOR S7	S1 XOR S5 XOR S6 XOR S7	S0 XOR S2 XOR S6 XOR S7	S3 XOR S5 XOR S6 XOR S7	S0 XOR S4 XOR S6									

21) S multiply with 3D

Comments\bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	S0	S1	S2	S3	S4	S5	S6	S7								
x 3D	1	0	1	1	1	1	0	0								
	S0	S1	S2	S3	S4	S5	S6	S7								
		0	0	0	0	0	0	0	0							
			S0	S1	S2	S3	S4	S5	S6	S7						
				S0	S1	S2	S3	S4	S5	S6	S7					
					S0	S1	S2	S3	S4	S5	S6	S7				
						S0	S1	S2	S3	S4	S5	S6	S7			
							0	0	0	0	0	0	0	0		
								0	0	0	0	0	0	0	0	
Answer	S0	S1	S0 XOR S2	S0 XOR S1 XOR S3	S0 XOR S1 XOR S2 XOR S4	S0 XOR S1 XOR S2 XOR S3 XOR S5	S1 XOR S2 XOR S3 XOR S4 XOR S6	S2 XOR S3 XOR S4 XOR S5 XOR S7	S3 XOR S4 XOR S5 XOR S6	S4 XOR S5 XOR S6 XOR S7	S5 XOR S6 XOR S7	S6 XOR S7	S7			
Primitive					1	0	1	1	0	0	1	0	1			
XOR S7					S7		S7	S7			S7		S7			
	S0	S1	S0 XOR S2	S0 XOR S1 XOR S3	S0 XOR S1 XOR S2 XOR S4	S0 XOR S1 XOR S2 XOR S3 XOR S5	S1 XOR S2 XOR S3 XOR S4 XOR S6	S2 XOR S3 XOR S4 XOR S5 XOR S7	S3 XOR S4 XOR S5 XOR S6	S4 XOR S5 XOR S6 XOR S7	S5 XOR S6 XOR S7	S6 XOR S7				

					XOR S4 XOR S7	XOR S3 XOR S5	XOR S4 XOR S6 XOR S7	XOR S5	XOR S6	XOR S7							
Primitive				1	0	1	1	0	0	1	0	1					
XOR (S6 XOR S7)				S6 XOR S7		S6 XOR S7	S6 XOR S7			S6 XOR S7		S6 XOR S7					
	S0	S1	S0 XOR S2	S0 XOR S1 XOR S3 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S4 XOR S7	S0 XOR S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6	S2 XOR S3 XOR S4 XOR S5	S3 XOR S4 XOR S5 XOR S6	S4 XOR S5	S5 XOR S6						
Primitive			1	0	1	1	0	0	1	0	1						
XOR (S5 XOR S6)			S5 XOR S6		S5 XOR S6	S5 XOR S6			S5 XOR S6		S5 XOR S6						
	S0	S1	S0 XOR S2 XOR	S0 XOR S1 XOR	S0 XOR S1 XOR	S0 XOR S1 XOR	S1 XOR S2 XOR	S2 XOR S3 XOR	S3 XOR S4 XOR	S4 XOR S5							

			S5 XOR S6	S3 XOR S6 XOR S7	S2 XOR S4 XOR S5 XOR S6 XOR S7	S2 XOR S3 XOR S7	S3 XOR S4	S4 XOR S5									
Primitive		1	0	1	1	0	0	1	0	1							
XOR (S4 XOR S5)		S4 XOR S5		S4 XOR S5	S4 XOR S5			S4 XOR S5		S4 XOR S5							
	S0	S1 XOR S4 XOR S5	S0 XOR S2 XOR S5 XOR S6	S0 XOR S1 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S2 XOR S3 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S3 XOR S4	S2 XOR S3 XOR S4	S3 XOR S4								
Primitive	1	0	1	1	0	0	1	0	1								
XOR (S3 XOR S4)	S3 XOR S4		S3 XOR S4	S3 XOR S4			S3 XOR S4		S3 XOR S4								
Final Answer	S0	S1	S0	S0	S0	S0	S1	S2									

(Recovered)	XOR S3 XOR S4	XOR S4 XOR S5	XOR S2 XOR S3 XOR S4 XOR S5 XOR S6	XOR S1 XOR S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	XOR S1 XOR S2 XOR S3 XOR S6 XOR S7	XOR S1 XOR S2 XOR S3 XOR S7	XOR S2	XOR S3										

22) S multiply with 19

Comments\bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	S0	S1	S2	S3	S4	S5	S6	S7								
x 19	1	0	0	1	1	0	0	0								
	S0	S1	S2	S3	S4	S5	S6	S7								
		0	0	0	0	0	0	0	0							
			0	0	0	0	0	0	0	0						
				S0	S1	S2	S3	S4	S5	S6	S7					
					S0	S1	S2	S3	S4	S5	S6	S7				
						0	0	0	0	0	0	0	0			
							0	0	0	0	0	0	0	0		
								0	0	0	0	0	0	0	0	
Answer	S0	S1	S2	S0 XOR S3	S0 XOR S1 XOR S4	S1 XOR S2 XOR S5	S2 XOR S3 XOR S6	S3 XOR S4 XOR S7	S4 XOR S5	S5 XOR S6	S6 XOR S7	S7				
Primitive				1	0	1	1	0	0	1	0	1				
XOR S7				S7		S7	S7			S7		S7				
	S0	S1	S2	S0 XOR S3 XOR S7	S0 XOR S1 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S4 XOR S5 XOR S6 XOR S7	S2 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S3 XOR S4 XOR S5 XOR S6 XOR S7	S4 XOR S5	S5 XOR S6 XOR S7	S6 XOR S7					
Primitive			1	0	1	1	0	0	1	0	1					

XOR (S6 XOR S7)			S6 XOR S7		S6 XOR S7	S6 XOR S7			S6 XOR S7		S6 XOR S7					
	S0	S1	S2 XOR S6 XOR S7	S0 XOR S3 XOR S7	S0 XOR S1 XOR S4 XOR S6 XOR S7	S1 XOR S2 XOR S5 XOR S6	S2 XOR S3 XOR S6 XOR S7	S3 XOR S4 XOR S7	S4 XOR S5 XOR S6 XOR S7	S5 XOR S6 XOR S7						
Primitive		1	0	1	1	0	0	1	0	1						
XOR (S5 XOR S6 XOR S7)		S5 XOR S6 XOR S7		S5 XOR S6 XOR S7	S5 XOR S6 XOR S7			S5 XOR S6 XOR S7		S5 XOR S6 XOR S7						
	S0	S1 XOR S5 XOR S6 XOR S7	S2 XOR S6 XOR S7	S0 XOR S3 XOR S5 XOR S7	S0 XOR S1 XOR S4 XOR S5 XOR S6	S1 XOR S2 XOR S5 XOR S6	S2 XOR S3 XOR S6 XOR S7	S3 XOR S4 XOR S5 XOR S6	S4 XOR S5 XOR S6 XOR S7							
Primitive	1	0	1	1	0	0	1	0	1							
XOR (S4 XOR S5 XOR S6 XOR S7)	S4 XOR S5 XOR S6		S4 XOR S5 XOR S6	S4 XOR S5 XOR S6			S4 XOR S5 XOR S6		S4 XOR S5 XOR S6							

	XOR S7		XOR S7	XOR S7			XOR S7		XOR S7							
Final Answer (Recovered)	S0 XOR S4 XOR S5 XOR S6 XOR S7	S1 XOR S5 XOR S6 XOR S7	S2 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S0 XOR S1 XOR S3 XOR S4 XOR S5 XOR S6 XOR S7	S1 XOR S2 XOR S4 XOR S5 XOR S6 XOR S7	S2 XOR S3 XOR S5 XOR S6 XOR S7	S3 XOR S4 XOR S5 XOR S6 XOR S7								

23) S multiply with 03

Comments\bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	S0	S1	S2	S3	S4	S5	S6	S7								
x 03	1	1	0	0	0	0	0	0								
	S0	S1 S0	S2 S1 0	S3 S2 0 0	S4 S3 0 0 0	S5 S4 0 0 0 0	S6 S5 0 0 0 0 0	S7 S6 0 0 0 0 0 0	S7 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0	0 0 0 0 0	0 0 0 0	0 0 0	0 0	0
Answer	S0	S0 XOR S1	S1 XOR S2	S2 XOR S3	S3 XOR S4	S4 XOR S5	S5 XOR S6	S6 XOR S7	S7							
Primitive	1	0	1	1	0	0	1	0	1							
XOR S7	S7		S7	S7			S7		S7							
	S0 XOR S7	S0 XOR S1	S1 XOR S2 XOR S7	S2 XOR S3 XOR S7	S3 XOR S4	S4 XOR S5	S5 XOR S6 XOR S7	S6 XOR S7								

APPENDIX C: SOURCE CODE FOR DESIGN 2

1. ADDER01

```
-------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja        ---
--- ID :  1669                                ---
--- Component : This component is a 1 bit adder. ---
--- Version : 1.0 - Beta                      ---
-------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity adder01 is
  port (dataa, datab, cin: in std_logic;
        result, cout: out std_logic);
end adder01;

architecture dataflow of adder01 is
begin
  result <= dataa xor datab xor cin;
  cout <= (dataa and datab) or (cin and (dataa or datab));
end dataflow;
```

2. ADDER32

```
-------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja        ---
--- ID :  1669                                ---
--- Component : This component is a 32 bit adder. ---
--- Version : 1.0 - Beta                      ---
-------------------------------------------------


library ieee ;
use ieee.std_logic_1164.all ;

entity adder32 is
   port( dataa, datab : in  std_logic_vector(31 downto 0) ;
         cout : out std_logic ;
         cin  : in std_logic ;
         result : out std_logic_vector(31 downto 0)) ;
end adder32;

architecture behavior of adder32 is
   component adder01
     port( dataa    : in std_logic ;
           datab    : in std_logic ;
           cin  : in std_logic ;
           result   : out std_logic ;
           cout : out std_logic ) ;
   end component;
   signal  c: std_logic_vector(31 downto 1) ;

begin
   g: for i in 0 to 31 generate

      g0: if (i = 0) generate
```



```
          x0: adder01 port map(dataa(0), datab(0), cin, result(0),
c(1)) ;
      end generate g0 ;

      g1: if ((i > 0) and (i < 31)) generate
          x0: adder01 port map(dataa(i), datab(i), c(i), result(i),
c(i+1)) ;
      end generate g1 ;

      g2: if (i = 31) generate
          x0: adder01 port map(dataa(31), datab(31), c(31),
result(31), cout) ;
      end generate g2 ;
   end generate g ;
end behavior;
```

3. CIPHERTEXT

```
-------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja              ---
--- ID :  1669                                      ---
--- Component : This component is a ciphertext.     ---
--- output part of the twofish enc/dec system:      ---
--  output whitening and result latching            ---
--- Version : 1.0 - Beta                            ---
-------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

--Ciphertext: output part of the twofish enc/dec system: output
whitening and result latching

entity ciphertext is
port( K4,K5,K6,K7 : in std_logic_vector(31 downto 0); --input for K-
Subkeys
      line0,line1,line2,line3 : in std_logic_vector(31 downto 0); --
inputs derived from 16 iterations in core component
      clk : in std_logic; --Clock signal
      reset : in std_logic;
      latch_ciphertext : in std_logic; --signal enabling latching of
results into result register for MS 64 bits
      ciphertext : out std_logic_vector(127 downto 0)); --Enc/Dec
result
end ciphertext;

--Start of ciphertext structure description
architecture mixed OF ciphertext is

component xor_2x32
port (A : in std_logic_vector(31 downto 0);
      B : in std_logic_vector(31 downto 0);
      Result : out std_logic_vector(31 downto 0));
end component;

component reg32
port(clock,clr,enable : in  std_logic;
     data : in std_logic_vector (31 downto 0);
     q : out std_logic_vector (31 downto 0));
```

```
end component;

signal C0, C1, C2, C3: std_logic_vector(31 downto 0); --Register
output signal
signal K4xorL0, K5xorL1,K6xorL2, K7xorL3: std_logic_vector(31 downto
0); --Resulting signal for XOR of inputs with different K-Subkeys
begin

-- Writing result of enc/dec to output register for LS 32-bit vector
C0_reg: reg32
port map (data => K4xorL0,
          clock => clk,
          clr => reset,
          enable => latch_ciphertext,
          q => C0);

-- Writing result of enc/dec to output register for second LS 32-bit
vector
C1_reg: reg32
port map (data => K5xorL1,
          clock => clk,
          clr => reset,
          enable => latch_ciphertext,
          q => C1);

-- Writing result of enc/dec to output register for second MS 32-bit
vector
C2_reg: reg32
port map (data => K6xorL2,
          clock => clk,
          clr => reset,
          enable => latch_ciphertext,
          q => C2);

-- Writing result of enc/dec to output register for MS 32-bit vector
C3_reg: reg32
port map (data => K7xorL3,
          clock => clk,
          clr => reset,
          enable => latch_ciphertext,
          q => C3);
--
-- output whitening
--

-- XOR LS 32-bit vector with K-Subkey K4
L0K4_xor : xor_2x32
port map (A => line0,
          B => K4,
          Result => K4xorL0);

-- XOR second LS 32-bit vector with K-Subkey K5
L1K5_xor : xor_2x32
port map (A => line1,
          B => K5,
          Result => K5xorL1);

-- XOR second MS 32-bit vector with K-Subkey K6
L2K6_xor : xor_2x32
port map (A => line2,
          B => K6,
          Result => K6xorL2);

-- XOR MS 32-bit vector with K-Subkey K7
L3K7_xor : xor_2x32
port map (A => line3,
          B => K7,
          Result => K7xorL3);

--Concatenating and assigning output its value from signal
output: process(C3,C2,C1,C0)
begin
ciphertext <= C3 & C2 & C1 & C0;
end process output;
end mixed;
```

4. CLEARTEXT

```
-------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                ---
--- ID :  1669                                        ---
--- Component : This component is the cleartext.      ---
---               Input Whitening                     ---
--- Version : 1.0 - Beta                              ---
-------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

--Building block of cleartext
entity cleartext is
port(input : in std_logic_vector(127 downto 0); --data to encrypt
input
      clk : in std_logic; --clock signal
      reset : in std_logic;
      latch_cleartext : in std_logic; --enable input register
latching
      K0,K1,K2,K3 : in std_logic_vector(31 downto 0); --K-Subkeys
input
      P0xorK0 : out std_logic_vector(31 downto 0); --LS 32-bit vector
whitening
      P1xorK1 : out std_logic_vector(31 downto 0); --Second LS 32-bit
vector whitening
      P2xorK2 : out std_logic_vector(31 downto 0); --Second to MS 32-
bit vector whitening
      P3xorK3 : out std_logic_vector(31 downto 0)); --MS 32-bit vecto
whitening
end cleartext;

--Start cleartext structure description
architecture mixed OF cleartext is

component xor_2x32
port(A : in std_logic_vector(31 downto 0);
     B : in std_logic_vector(31 downto 0);
     Result : out std_logic_vector(31 downto 0));
end component;

component reg128
```

```
port(clock,clr,enable : in  std_logic;
     data : in std_logic_vector (127 downto 0);
     q : out std_logic_vector (127 downto 0));
end component;


signal cleartext_out : std_logic_vector(127 downto 0); --output
signal
--Cleartext: input to enc/dec unit: latching input into register,
input whitening and S0 and S1 sub-key generation
begin

-- Latching input into register
cleartext_input: reg128
port map (data => input,
          clock => clk,
          clr => reset,
          enable => latch_cleartext,
          q => cleartext_out);

-- input whitening by XORing input to deifferent K-Subkeys
P0K0_xor : xor_2x32
port map (A => cleartext_out(31 downto 0),
          B => K0,
          Result => P0xorK0);
P1K1_xor : xor_2x32
port map (A => cleartext_out(63 downto 32),
          B => K1,
          Result => P1xorK1);
P2K2_xor : xor_2x32
port map (A => cleartext_out(95 downto 64),
          B => K2,
          Result => P2xorK2);
P3K3_xor : xor_2x32
port map (A => cleartext_out(127 downto 96),
          B => K3,
          Result => P3xorK3);
end mixed;
```

5. CNTR

```
-------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja        ---
--- ID :  1669                                ---
--- Component : This component is a counter   ---
---             It drives through the sequence ---
---             of encryption/decryption      ---
--- Version : 1.0 - Beta                      ---
-------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity cntr is
port (clk : in std_logic;
      enable : in std_logic;
      clr : in std_logic;
      encrypt : in std_logic;
      out_cntr0 : out integer range 0 to 39;
      out_cntr1 : out integer range 0 to 39;
      out_cntr2 : out integer range 0 to 39;
      out_cntr3 : out integer range 0 to 39);
end cntr;

architecture behavior of cntr is
signal cnt_sig0,
            cnt_sig1,
            cnt_sig2,
            cnt_sig3 : integer range 0 to 39;


begin
process (clk)
begin
if (clk'EVENT AND clk = '1') then
     if clr = '1' then
          if encrypt = '1' then
               cnt_sig0 <= 0;
               cnt_sig1 <= 1;
               cnt_sig2 <= 2;
               cnt_sig3 <= 3;
          else
               cnt_sig0 <= 4;
               cnt_sig1 <= 5;
               cnt_sig2 <= 6;
               cnt_sig3 <= 7;
          end if;
     elsif enable = '1' then
          if encrypt = '1' then
               if cnt_sig0 = 0 and cnt_sig1 = 1 and cnt_sig2 = 2
and cnt_sig3 = 3  then
                    cnt_sig0 <= 8;
                    cnt_sig1 <= 9;
                    cnt_sig2 <= 0;
                    cnt_sig3 <= 0;

               elsif cnt_sig0 = 38 and cnt_sig1 = 39 and cnt_sig2
= 0 and cnt_sig3 = 0  then
                    cnt_sig0 <= 4;
                    cnt_sig1 <= 5;
                    cnt_sig2 <= 6;
                    cnt_sig3 <= 7;

               elsif cnt_sig0 = 4 and cnt_sig1 = 5 and cnt_sig2 =
6 and cnt_sig3 = 7  then
                    cnt_sig0 <= 0;
                    cnt_sig1 <= 1;
                    cnt_sig2 <= 2;
                    cnt_sig3 <= 3;
               else
                    cnt_sig0 <= cnt_sig0 + 2;
                    cnt_sig1 <= cnt_sig1 + 2;
                    cnt_sig2 <= 0;
                    cnt_sig3 <= 0;
               end if;
          else
               if cnt_sig0 = 4 and cnt_sig1 = 5 and cnt_sig2 = 6
and cnt_sig3 = 7  then
                    cnt_sig0 <= 38;
```

```
                        cnt_sig1 <= 39;
                        cnt_sig2 <= 0;
                        cnt_sig3 <= 0;

                  elsif cnt_sig0 = 8 and cnt_sig1 = 9 and cnt_sig2 =
0 and cnt_sig3 = 0 then
                        cnt_sig0 <= 0;
                        cnt_sig1 <= 1;
                        cnt_sig2 <= 2;
                        cnt_sig3 <= 3;

                  elsif cnt_sig0 = 0 and cnt_sig1 = 1 and cnt_sig2 =
2 and cnt_sig3 = 3 then
                        cnt_sig0 <= 4;
                        cnt_sig1 <= 5;
                        cnt_sig2 <= 6;
                        cnt_sig3 <= 7;

                  else
                        cnt_sig0 <= cnt_sig0 - 2;
                        cnt_sig1 <= cnt_sig1 - 2;
                        cnt_sig2 <= 0;
                        cnt_sig3 <= 0;

                  end if;
            end if;
      end if;
end if;
end process;
out_cntr0 <= cnt_sig0;
out_cntr1 <= cnt_sig1;
out_cntr2 <= cnt_sig2;
out_cntr3 <= cnt_sig3;
end behavior;
```

6. CONTROL

```
-------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja              ---
--- ID :  1669                                      ---
--- Component : This component is the controller.   ---
--- Version : 1.0 - Beta                            ---
-------------------------------------------------------

 --Library Declaration
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

--Control: Controller module used to synchronize enc/dec different
steps
entity control is
port( pre_param_even : in std_logic_vector(31 downto 0);
      pre_param_odd : in std_logic_vector(31 downto 0);
      clk : in std_logic;
      reset : in std_logic;
      ld_key : in std_logic;
      start : in std_logic;
      sel_enc : in std_logic;
      input_even_key : out std_logic_vector(31 downto 0); -- need to
expand to 4 values
      input_odd_key : out std_logic_vector(31 downto 0); -- need to
expand to 4 values
      input_even_plaintext : out std_logic_vector(31 downto 0); --
need to expand to 4 values
      input_odd_plaintext : out std_logic_vector(31 downto 0); --
need to expand to 4 values
      latch_cleartext : out std_logic;
      latch_key : out std_logic;
      sel_odd : out std_logic;
      sel_k : out std_logic;
      sel_test :out std_logic;
      param_source_is_result : out std_logic;
      load_reg_pre_even : out std_logic;
      load_reg_pre_odd : out std_logic;
      load_reg_post_even : out std_logic;
      load_reg_post_odd : out std_logic;
      latch_ciphertext : out std_logic;
      ctrl_encrypt : out std_logic;
      idle : out std_logic);
end control;

--Building block of controller
architecture mix of control is
--
--component DECLARATION
--
--counter module
component cntr
port (clk : in std_logic;
      enable : in std_logic;
      clr : in std_logic;
      encrypt : in std_logic;
      out_cntr0 : out integer range 0 to 39;
      out_cntr1 : out integer range 0 to 39;
      out_cntr2 : out integer range 0 to 39;
      out_cntr3 : out integer range 0 to 39 );
end component;

--
--type DECLARATION
--
type state_type0 is (my_idle, load_keyA, compute_K0,compute_K4,
                                          compute_K2r_8);

type state_type1 is (my_idle, load_keyA,compute_K1,compute_K5,
                                         compute_K2r_9);

type state_type2 is (my_idle, load_keyA, compute_K2, compute_K6,
                                          compute_even);

type state_type3 is (my_idle, load_keyA,compute_K3, compute_K7,
                                         compute_odd);

--signal DECLARATION
--
signal state0 : state_type0;
signal state1 : state_type1;
signal state2 : state_type2;
signal state3 : state_type3;
```

```
signal
      cnt_enbl : std_logic;

signal
      out_count0,
      out_count1,
      out_count2,
      out_count3 : std_logic_vector(7 downto 0);

signal
      out_count_int0,
      out_count_int1,
      out_count_int2,
      out_count_int3    : integer range 0 to 39;

signal
      load_encrypt : std_logic;
signal
      reset_cntr,
      ctrl_enc_sig : std_logic;

--Start of controller structure description
begin
ctrl_encrypt <= ctrl_enc_sig;


cntr1 : cntr
port map( clk => clk,
          enable => cnt_enbl,
          clr => reset_cntr,
          encrypt => sel_enc,
          out_cntr0 => out_count_int0,
          out_cntr1 => out_count_int1,
          out_cntr2 => out_count_int2,
          out_cntr3 => out_count_int3);


-- EVEN KEY

process (clk,reset)
begin
if reset ='1' then
      state0 <= my_idle;
elsif clk'EVENT AND clk = '1' then
      case state0 is
            when my_idle =>
                  if ld_key = '1' then
                        state0 <= load_keyA; --'LoadKey' signal makes
the controller to latch the key material from the 128 bit input port
                  elsif start = '1' then -- 'start' signal puts the
controller in the encryption or decryption mode
                        state0 <= compute_K0;
                  else state0 <= my_idle;
                  end if;
            when compute_K0 =>
                  state0 <= compute_K2r_8;
            when compute_K2r_8 =>
                  if ( ((out_count0 = "00100110") AND (ctrl_enc_sig =
'1'))       -- for encryption
                        OR ((out_count0 = "00001000") AND (ctrl_enc_sig =
'0')) ) then  -- for decryption
                              state0 <= compute_K4;
                        else
                              state0 <= compute_K2r_8;
                        end if;
                  when load_keyA =>
                        state0 <= my_idle;
                  when compute_K4 =>
                        state0 <= my_idle;
      end case;
end if;
end process;

-- ODD KEY

process (clk,reset)
begin
if reset ='1' then
      state1 <= my_idle;
elsif clk'EVENT AND clk = '1' then
      case state1 is
            when my_idle =>
                  if ld_key = '1' then
                        state1 <= load_keyA; --'LoadKey' signal makes
the controller to latch the key material from the 128 bit input port
                  elsif start = '1' then -- 'start' signal puts the
controller in the encryption or decryption mode
                        state1 <= compute_K1;
                  else state1 <= my_idle;
                  end if;
            when compute_K1 =>
                  state1 <= compute_K2r_9;
            when compute_K2r_9 =>
                  if ( ((out_count0 = "00100110") AND (ctrl_enc_sig =
'1'))       -- for encryption
                        OR ((out_count0 = "00001000") AND (ctrl_enc_sig =
'0')) ) then  -- for decryption
                              state1 <= compute_K5;
                        else
                              state1 <= compute_K2r_9;
                        end if;
                  when load_keyA =>
                        state1 <= my_idle;
                  when compute_K5 =>
                        state1 <= my_idle;
                  end case;
end if;
end process;


-- EVEN PLAINTEXT

process (clk,reset)
begin
if reset ='1' then
      state2 <= my_idle;
elsif clk'EVENT AND clk = '1' then
      case state2 is
            when my_idle =>
                  if ld_key = '1' then
```

```
                        state2 <= load_keyA; --'LoadKey' signal makes
the controller to latch the key material from the 128 bit input port
                  elsif start = '1' then -- 'start' signal puts the
controller in the encryption or decryption mode
                        state2 <= compute_K2;
                  else state2 <= my_idle;
                  end if;
            when compute_K2 =>
            state2 <= compute_even;
            when compute_even =>
                  if ( ((out_count0 = "00100110") AND (ctrl_enc_sig =
'1'))       -- for encryption
                  OR ((out_count0 = "00001000") AND (ctrl_enc_sig =
'0')) ) then  -- for decryption
                        state2 <= compute_K6;
                  else
                        state2 <= compute_even;
                  end if;
            when load_keyA =>
                  state2 <= my_idle;
            when compute_K6 =>
                  state2 <= my_idle;

      end case;
end if;
end process;


-- ODD PLAINTEXT


process (clk,reset)
begin
if reset ='1' then
      state3 <= my_idle;
elsif clk'EVENT AND clk = '1' then
      case state3 is
            when my_idle =>
                  if ld_key = '1' then
                        state3 <= load_keyA; --'LoadKey' signal makes
the controller to latch the key material from the 128 bit input port
                  elsif start = '1' then -- 'start' signal puts the
controller in the encryption or decryption mode
                        state3 <= compute_K3;
                  else state3 <= my_idle;
                  end if;
            when compute_K3 =>
                        state3 <= compute_odd;
            when compute_odd =>
                  if ( ((out_count0 = "00100110") AND (ctrl_enc_sig =
'1'))       -- for encryption
                  OR ((out_count0 = "00001000") AND (ctrl_enc_sig =
'0')) ) then  -- for decryption
                        state3 <= compute_K7;
                  else
                        state3 <= compute_odd;
                  end if;
            when load_keyA =>
                  state3 <= my_idle;
            when compute_K7 =>
                  state3 <= my_idle;
      end case;
end if;
end process;


--EVEN KEY -- INPUT_EVEN_KEY
with state0 select input_even_key <=
(out_count0 & out_count0 & out_count0 & out_count0) when my_idle,
(out_count0 & out_count0 & out_count0 & out_count0) when load_KeyA,
(out_count0 & out_count0 & out_count0 & out_count0) when compute_K0,
(out_count0 & out_count0 & out_count0 & out_count0) when compute_K4,
(out_count0 & out_count0 & out_count0 & out_count0) when
compute_K2r_8;


-- ODD KEY -- INPUT_ODD_KEY
with state1 select input_odd_key <=
(out_count1 & out_count1 & out_count1 & out_count1) when my_idle,
(out_count1 & out_count1 & out_count1 & out_count1) when load_KeyA,
(out_count1 & out_count1 & out_count1 & out_count1) when compute_K1,
(out_count1 & out_count1 & out_count1 & out_count1) when compute_K5,
(out_count1 & out_count1 & out_count1 & out_count1) when
compute_K2r_9;


--EVEN PLAINTEXT -- INPUT_EVEN_PLAINTEXT
with state2 select input_even_plaintext <=
(out_count2 & out_count2 & out_count2 & out_count2) when my_idle,
(out_count2 & out_count2 & out_count2 & out_count2) when load_KeyA,
(out_count2 & out_count2 & out_count2 & out_count2) when compute_K2,
(out_count2 & out_count2 & out_count2 & out_count2) when compute_K6,
pre_param_even when compute_even;


--ODD PLAINTEXT -- INPUT_ODD_PLAINTEXT
with state3 select input_odd_plaintext <=
(out_count3 & out_count3 & out_count3 & out_count3) when my_idle,
(out_count3 & out_count3 & out_count3 & out_count3) when load_KeyA,
(out_count3 & out_count3 & out_count3 & out_count3) when compute_K3,
(out_count3 & out_count3 & out_count3 & out_count3) when compute_K7,
pre_param_odd when compute_odd;



--EVEN KEY -- LATCH_KEY
with state0 select latch_key <= '0' when my_idle,
                                 '1' when load_KeyA,
                                 '0' when compute_K0,
                                 '0' when compute_K4,
                                 '0' when compute_K2r_8;

--EVEN KEY -- RESET_CNTR
with state0 select reset_cntr <= '1' when my_idle,
                                 '0' when load_KeyA,
                                 '0' when compute_K0,
                                 '0' when compute_K4,
                                 '0' when compute_K2r_8;
```

```
--EVEN KEY -- LATCH_CLEARTEXT
with state0 select latch_cleartext <= '1' when my_idle,
                                      '0' when load_KeyA,
                                      '0' when compute_K0,
                                      '0' when compute_K4,
                                      '0' when compute_K2r_8;

--ODD PLAINTEXT -- SEL_ODD

with state3 select sel_odd <=   '0' when my_idle,
                                '0' when load_KeyA,
                                '1' when compute_K3,
                                '1' when compute_K7,
                                '1' when compute_odd;

--ODD PLAINTEXT -- SEL_K

with state3 select  sel_k <=    '0' when my_idle,
                                '0' when load_KeyA,
                                '1' when compute_K3,
                                '1' when compute_K7,
                                '0' when compute_odd;


--ODD PLAINTEXT -- SEL_TEST

with state3 select
          sel_test <=    '0' when my_idle,
                                '0' when load_KeyA,
                                '1' when compute_K3,
                                '1' when compute_K7,
                                '0' when compute_odd;


--EVEN KEY -- PARAM_SOURCE_IS_RESULT
with state0 select param_source_is_result <= '0' when my_idle,
                                             '0' when load_KeyA,
                                             '0' when compute_K0,
                                             '1' when compute_K4,
                                             '1' when compute_K2r_8;


--EVEN KEY  -- LOAD_REG_PRE_EVEN
with state0 select load_reg_pre_even <= '0' when my_idle,
                                        '0' when load_KeyA,
                                        '1' when compute_K0,
                                        '0' when compute_K4,
                                        '1' when compute_K2r_8;


-- ODD KEY -- LOAD_REG_PRE_ODD

with state1 select load_reg_pre_odd <= '0' when my_idle,
                                       '0' when load_KeyA,
                                       '1' when compute_K1,
                                       '0' when compute_K5,
                                       '1' when compute_K2r_9;
```

```
--EVEN PLAINTEXT -- LOAD_REG_POST

with state2 select load_reg_post_even <= '0' when my_idle,
                                         '0' when load_KeyA,
                                         '1' when compute_K2,
                                         '0' when compute_K6,
                                         '1' when compute_even;


--ODD PLAINTEXT -- LOAD_REG_POST_ODD

with state3 select load_reg_post_odd <= '0' when my_idle,
                                        '0' when load_KeyA,
                                        '1' when compute_K3,
                                        '0' when compute_K7,
                                        '1' when compute_odd;


--EVEN PLAINTEXT -- LATCH_CIPHER_MS_NEXTCC

with state2 select latch_ciphertext <=  '0' when my_idle,
                                        '0' when load_KeyA,
                                        '0' when compute_K2,
                                        '1' when compute_K6,
                                        '0' when compute_even;

--EVEN KEY -- LOAD_ENCRYPT
with state0 select load_encrypt <= '1' when my_idle,
                                   '0' when load_KeyA,
                                   '0' when compute_K0,
                                   '0' when compute_K4,
                                   '0' when compute_K2r_8;


--EVEN KEY -- IDLE

with state0 select idle <= '1' when my_idle,
                           '0' when load_KeyA,
                           '0' when compute_K0,
                           '0' when compute_K4,
                           '0' when compute_K2r_8;


--EVEN KEY -- CNT_ENABLE

with state0 select cnt_enbl <= '0' when my_idle,
                               '0' when load_KeyA,
                               '1' when compute_K0,
                               '1' when compute_K4,
                               '1' when compute_K2r_8;


sel_encrypt_reg: process(load_encrypt)
begin

      if (load_encrypt = '1') then
            ctrl_enc_sig <= sel_enc;
      end if;

end process;
```

```
out_count0 <= conv_std_logic_vector(out_count_int0,8);
out_count1 <= conv_std_logic_vector(out_count_int1,8);
out_count2 <= conv_std_logic_vector(out_count_int2,8);
out_count3 <= conv_std_logic_vector(out_count_int3,8);


end mix;
```

7. CORE

```
-------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja          ---
--- ID :  1669                                  ---
--- Component : This component is the main core. ---
--- Version : 1.0 - Beta                        ---
-------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

-- External interface of twofish cipher implementation
entity core is
port(inport : in std_logic_vector(127 downto 0); --input from
cleartext entity
      inkey : in std_logic_vector (127 downto 0); -- input from
keymodule entity
      clk : in std_logic; --Clock signal
      reset : in std_logic;
      usr_ld_key : in std_logic; --Usr requests load key
      usr_start : in std_logic; --Usr requests start
      usr_encrypt : in std_logic; --Usr requests encrypt
      idle : out std_logic; --Device is idle
      outCiphertext : out std_logic_vector(127 downto 0));--output
after one iteration through enc/dec
end core;

architecture mixed of core is

component little_endian_converter
port (big_endian : in std_logic_vector(127 downto 0);
      little_endian : out std_logic_vector(127 downto 0));
end component;

component control
port( pre_param_even : in std_logic_vector(31 downto 0);
      pre_param_odd : in std_logic_vector(31 downto 0);
      clk : in std_logic;
      reset : in std_logic;
      ld_key : in std_logic;
      start : in std_logic;
      sel_enc : in std_logic;
      input_even_key : out std_logic_vector(31 downto 0); -- need to
expand to 4 values
      input_odd_key : out std_logic_vector(31 downto 0); -- need to
expand to 4 values
      input_even_plaintext : out std_logic_vector(31 downto 0); --
need to expand to 4 values
      input_odd_plaintext : out std_logic_vector(31 downto 0); --
need to expand to 4 values
      latch_cleartext : out std_logic;
      latch_key : out std_logic;
      sel_odd : out std_logic;
      sel_k : out std_logic;
      sel_test : out std_logic;
      param_source_is_result : out std_logic;
      load_reg_pre_even : out std_logic;
      load_reg_pre_odd : out std_logic;
      load_reg_post_even : out std_logic;
      load_reg_post_odd : out std_logic;
      latch_ciphertext : out std_logic;
      ctrl_encrypt : out std_logic;
      idle : out std_logic);
end component;

component keymodule
port (key : in std_logic_vector(127 downto 0);
      latch_key : in std_logic;
      clk : in std_logic;
      reset : in std_logic;
      S0,S1 : out std_logic_vector(31 downto 0);
      keyout : out std_logic_vector(127 downto 0));
end component;

component evenkeygenerator
port (input : in std_logic_vector(31 downto 0);
      in_S0 : in std_logic_vector(31 downto 0); --S0 for S-boxes in
h_fctn module
      in_S1 : in std_logic_vector(31 downto 0); --S1 for S-boxes in
h_fctn module
      clk : in std_logic;
      reset : in std_logic;
      outevenkey : out std_logic_vector(31 downto 0)); --output of
main on odd path
end component;

component oddkeygenerator
port (input : in std_logic_vector(31 downto 0);
      in_S0 : in std_logic_vector(31 downto 0); --S0 for S-boxes in
h_fctn module
      in_S1 : in std_logic_vector(31 downto 0); --S1 for S-boxes in
h_fctn module
      clk : in std_logic;
      reset : in std_logic;
      outoddkey : out std_logic_vector(31 downto 0)); --output of
main odd key
end component;

component evenplaintextgenerator
port (input : in std_logic_vector(31 downto 0);
      in_S0 : in std_logic_vector(31 downto 0); --S0 for S-boxes in
h_fctn module
      in_S1 : in std_logic_vector(31 downto 0); --S1 for S-boxes in
h_fctn module
      clk : in std_logic;
      reset : in std_logic;
      outevenplaintext : out std_logic_vector(31 downto 0)); --output
of main on odd path
end component;
```

```
component oddplaintextgenerator
port (input : in std_logic_vector(31 downto 0);
      in_S0 : in std_logic_vector(31 downto 0); --S0 for S-boxes in
h_fctn module
      in_S1 : in std_logic_vector(31 downto 0); --S1 for S-boxes in
h_fctn module
      clk : in std_logic;
      reset : in std_logic;
      sel_odd : in std_logic; --input selecting btw even path or odd
path
      sel_k : in std_logic; --input selecting btw calculating a key
or an encryption/decryption
      outoddplaintext : out std_logic_vector(31 downto 0)); --output
of main on odd path
end component;

component rol9_32
port (data : in std_logic_vector(31 downto 0);
      q : out std_logic_vector(31 downto 0));
end component;

--rotate left 8 bit
component rol8_32
port (data : in std_logic_vector(31 downto 0);
      q : out std_logic_vector(31 downto 0));
end component;

component mux_2x32
port(sel : in std_logic;
           in_0 : in std_logic_vector(31 downto 0);
           in_1 : in std_logic_vector(31 downto 0);
           output : out std_logic_vector(31 downto 0));
end component;

--pht modules
component pht
port (A_in : in std_logic_vector(31 downto 0);
      B_in : in std_logic_vector(31 downto 0);
      A_out : out std_logic_vector(31 downto 0);
      B_out : out std_logic_vector(31 downto 0));
end component;

component adder32
port (dataa, datab : in  std_logic_vector(31 downto 0) ;
      cout : out std_logic ;
      cin  : in std_logic:= '0';
      result : out std_logic_vector(31 downto 0)) ;
end component;

component reg32
port(clock,clr,enable : in  std_logic;
       data : in std_logic_vector (31 downto 0);
       q : out std_logic_vector (31 downto 0));
end component;

-- selection of mode: encrypt/decrypt
component opselect
port( encrypt : in std_logic;
      F_fct_A : in std_logic_vector(31 downto 0);
      F_fct_B : in std_logic_vector(31 downto 0);
      in_A : in std_logic_vector(31 downto 0);
      in_B : in std_logic_vector(31 downto 0);
      out_A : out std_logic_vector(31 downto 0);
      out_B : out std_logic_vector(31 downto 0));
end component;

--cleartext module
component cleartext
port (input : in std_logic_vector(127 downto 0);
      clk : in std_logic;
      reset : in std_logic;
      latch_cleartext : in std_logic;
      K0,K1,K2,K3 : in std_logic_vector(31 downto 0);
      P0xorK0 : out std_logic_vector(31 downto 0);
      P1xorK1 : out std_logic_vector(31 downto 0);
      P2xorK2 : out std_logic_vector(31 downto 0);
      P3xorK3 : out std_logic_vector(31 downto 0));
end component;

--ciphetext module
component ciphertext
port (K4,K5,K6,K7 : in std_logic_vector(31 downto 0);
      line0,line1,line2,line3 : in std_logic_vector(31 downto 0);
      clk : in std_logic;
      reset : in std_logic;
      latch_ciphertext : in std_logic;
      ciphertext : out std_logic_vector(127 downto 0));
end component;

signal input_even_key,
            input_odd_key,
            input_even_plaintext,
            input_odd_plaintext,
            inputevenkeygenerator,
            outputevenkeygenerator,outputevenkeygenerator_temp,
            inputoddkeygenerator,
            outputoddkeygenerator,  outputoddkeygenerator_temp,
            inputevenplaintextgenerator,
            outputevenplaintextgenerator,
            inputoddplaintextgenerator,
            outputoddplaintextgenerator,
            input_even_key_pht,
            input_odd_key_pht,
            output_even_key_pht,
            output_odd_key_pht,
            rotated_output_odd_key_pht,
            even_key,
            odd_key,
            input_even_key_pht_temp,
            input_odd_key_pht_temp,
            output_even_key_pht_temp,
            output_odd_key_pht_temp,
            rotated_output_odd_key_pht_temp,
            even_key_temp,
            odd_key_temp,
            output_even_plaintext_pht,
            output_odd_plaintext_pht,
            rotated_output_odd_plaintext_pht,
            output_odd_plaintext_mux,
            even_plaintext,   even_plaintext_temp,
```

```
            odd_plaintext_temp,
            odd_plaintext,
            P0xorK0, P1xorK1,
            P2xorK2, P3xorK3,
            S0, S1,
            f_out_even, f_out_odd,pre_param_even,
            pre_param_odd,post_param_even,
            post_param_odd,
            input_f,out_round_even,
            out_round_odd,source_pre_even_reg,
            source_pre_odd_reg,
            source_post_even_reg, source_post_odd_reg, even_S0,
            even_S1,odd_S0, odd_S1,
            f_in_S0, f_in_S1
           : std_logic_vector(31 downto 0);

signal
      key, input, output, outkey : std_logic_vector(127 downto 0);

signal
      latch_cleartext, latch_key,
      sel_odd, sel_k, sel_odd_k,sel_test,
      param_source_is_result,
      latch_ciphertext,
      ctrl_encrypt,
      load_reg_pre_even, load_reg_pre_odd,
      load_reg_post_even, load_reg_post_odd : std_logic;

begin

--little endian convertion
little_endian_input: little_endian_converter
port map ( big_endian => inport,
           little_endian => input);

little_endian_key: little_endian_converter
port map ( big_endian => inkey,
           little_endian => key);

little_endian_output: little_endian_converter
port map ( big_endian => output,
           little_endian => outCiphertext);

twofishController: control
port map ( pre_param_even => pre_param_even,
           pre_param_odd => pre_param_odd,
           clk => clk,
           reset => reset,
           ld_key => usr_ld_key,
           start => usr_start,
           sel_enc => usr_encrypt,
           input_even_key => input_even_key,
           input_odd_key => input_odd_key,
           input_even_plaintext => input_even_plaintext,
           input_odd_plaintext => input_odd_plaintext,
           latch_cleartext => latch_cleartext,
           latch_key => latch_key,
           sel_odd => sel_odd,
           sel_k => sel_k,
           sel_test =>sel_test,
           param_source_is_result => param_source_is_result,
           load_reg_pre_even => load_reg_pre_even,
           load_reg_pre_odd => load_reg_pre_odd,
           load_reg_post_even => load_reg_post_even,
           load_reg_post_odd => load_reg_post_odd,
           latch_ciphertext => latch_ciphertext,
           ctrl_encrypt => ctrl_encrypt,
           idle => idle);

-- This modules holds the cleartext input module and
-- and prepares the parameters when given correct K values.
cleartext_module: cleartext
port map (input => input,
           clk => clk,
           reset => reset,
           latch_cleartext => latch_cleartext,
           K0 => even_key,
           K1 => odd_key,
           K2 => even_plaintext,
           K3 => odd_plaintext,
           P0xorK0 => P0xorK0,
           P1xorK1 => P1xorK1,
           P2xorK2 => P2xorK2,
           P3xorK3 => P3xorK3);

-- select the correct S input to modified F:
-- S if computing a result,
-- M if computing a k
selectSorM : process (sel_k,sel_odd,key,s0,s1)
begin
if (sel_k = '1') THEN
      if (sel_odd = '1') THEN
           -- computing a K
           -- S0 = M3, S1 = M1
           f_in_S0 <= outkey(127 downto 96);
           f_in_S1 <= outkey(63 downto 32);
      else
           -- computing a K
           -- S0 = M2, S1 = M0
           f_in_S0 <= outkey(95 downto 64);
           f_in_S1 <= outkey(31 downto 0);
      end if;
else
      -- computing a result
      -- S0 = S0, S1 = S1
      f_in_S0 <= S0;
      f_in_S1 <= S1;
end if;
end process;

inputevenkeygenerator <= input_even_key;
inputoddkeygenerator <= input_odd_key;
inputevenplaintextgenerator  <= input_even_plaintext;
inputoddplaintextgenerator <= input_odd_plaintext;

evenkeygen: evenkeygenerator
port map (input =>inputevenkeygenerator,
          in_S0 => outkey(95 downto 64),  --- must take into account
for key
          in_S1 => outkey(31 downto 0),  --- must take into account
for key
          clk => clk,
```

```
        reset => reset,
        outevenkey => outputevenkeygenerator);

oddkeygen: oddkeygenerator
port map (input =>inputoddkeygenerator,
          in_S0 => outkey(127 downto 96),   --- must take into
account for key
          in_S1 => outkey(63 downto 32),   --- must take into account
for key
          clk => clk,
          reset => reset,
          outoddkey => outputoddkeygenerator);


input_even_key_pht <= outputevenkeygenerator;
input_odd_key_pht <= outputoddkeygenerator;

--Pseudo Hadamard Transformation of signals for keys
pht_key : pht
port map (A_in => input_even_key_pht,
          B_in => input_odd_key_pht,
          A_out => output_even_key_pht,
          B_out => output_odd_key_pht);

--Rotating the output of the PHT on the odd path left nine times
rol9_out_pht_odd_key: rol9_32
port map (data => output_odd_key_pht ,
          q => rotated_output_odd_key_pht);

 --KEY VALUES
even_key <= output_even_key_pht;
odd_key <= rotated_output_odd_key_pht;

mux_sel_evenS0 : mux_2x32
port map(sel => sel_test,
         in_1 => outkey(95 downto 64),
         in_0 =>  f_in_S0,
         output => even_S0);

mux_sel_oddS0 : mux_2x32
port map(sel => sel_test,
         in_1 => outkey(127 downto 96),
         in_0 =>  f_in_S0,
         output => odd_S0);

mux_sel_evenS1 : mux_2x32
port map(sel => sel_test,
         in_1 =>  outkey(31 downto 0),
         in_0 => f_in_S1,
         output => even_S1);

mux_sel_oddS1 : mux_2x32
port map(sel => sel_test,
         in_1 => outkey(63 downto 32),
         in_0 => f_in_S1,
         output => odd_S1);

evenplaintextgen: evenkeygenerator
port map(input =>inputevenplaintextgenerator,
         in_S0 => even_S0,  -- must take into account for plaintext
          in_S1 => even_S1,  -- must take into account for plaintext
          clk => clk,
          reset => reset,
          outevenkey => outputevenplaintextgenerator);


oddplaintextgen: oddplaintextgenerator
port map(input =>inputoddplaintextgenerator,
         in_S0 => odd_S0,  -- must take into account for plaintext
         in_S1 => odd_S1,  -- must take into account for plaintext
         clk => clk,
         reset => reset,
         sel_odd => sel_odd,
         sel_k => sel_k,
         outoddplaintext => outputoddplaintextgenerator);

--Pseudo Hadamard Transformation of signals for plaintext
pht_plaintext : pht
port map (A_in => outputevenplaintextgenerator,
          B_in => outputoddplaintextgenerator,
          A_out => output_even_plaintext_pht,
          B_out => output_odd_plaintext_pht);

--Rotating the output of the PHT on the odd path left nine times
rol9_out_pht_odd: rol9_32
port map (data => output_odd_plaintext_pht ,
          q => rotated_output_odd_plaintext_pht);

sel_odd_k <= sel_odd AND sel_k; -- to choose between computation of a
key and plaintext

--Multiplex, on the odd side, btw. the output of the pht and the
output of the pht rol X9
mux_out : mux_2x32
port map(sel => sel_odd_k,
         in_1 => rotated_output_odd_plaintext_pht,
         in_0 => output_odd_plaintext_pht,
         output => output_odd_plaintext_mux);

even_plaintext <= output_even_plaintext_pht;
odd_plaintext <= output_odd_plaintext_mux;

-- 32-bit adder: even K + even result of h
add_even : adder32
port map (dataa => even_plaintext,
          datab => even_key,
          result => f_out_even);

-- 32-bit adder: odd K + odd result of h
add_odd : adder32
port map (dataa => odd_plaintext,
          datab => odd_key,
          result => f_out_odd);

-- final operation of round: encrypt or decrypt?
opsel : opselect
port map(encrypt => ctrl_encrypt,
         F_fct_A => f_out_even,
         F_fct_B => f_out_odd,
         in_A => post_param_even,
         in_B => post_param_odd,
         out_A => out_round_even,
```

```
        out_B => out_round_odd);

-- choose the source of next parameters...
choose_in_param_pre_even: process
(param_source_is_result,out_round_even,
 out_round_odd,pre_param_even,pre_param_odd,
 P0xorK0,P1xorK1,P2xorK2,P3xorK3)

begin
if (param_source_is_result = '1') THEN
      source_pre_even_reg <= out_round_even;
      source_pre_odd_reg <= out_round_odd;
      source_post_even_reg <= pre_param_even;
      source_post_odd_reg <= pre_param_odd;
else
      source_pre_even_reg <= P0xorK0;
      source_pre_odd_reg <= P1xorK1;
      source_post_even_reg <= P2xorK2;
      source_post_odd_reg <= P3xorK3;
end if;
end process ;

-- line 0: the parameter is used on even
-- values at the beginning of the round
-- (K0 in first round)
reg_pre_even_param_reg: reg32
port map (data => source_pre_even_reg,
          clock => clk,
          clr => reset,
          enable => load_reg_pre_even, -- helps to latch P0xorK0 or
out round even
            q => pre_param_even);

-- line 1: the parameter is used on odd
-- values at the beginning of the round
-- (K1 in first round)
reg_pre_odd_param_reg: reg32
port map (data => source_pre_odd_reg,
          clock => clk,
          clr => reset,
          enable => load_reg_pre_odd,     -- helps to latch P1xorK1
or out round odd
            q => pre_param_odd);

-- line 2: the parameter is used on even
-- values at the end of the round
-- (K2 in first round)
reg_post_even_param_reg: reg32
port map (data => source_post_even_reg,
          clock => clk,
          clr => reset,
          enable => load_reg_post_even,   -- helps to latch P2xorK2
or pre param even
          q => post_param_even);

-- line 3: the parameter is used on odd
-- values at the end of the round
-- (K3 in first round)
reg_post_odd_param_reg: reg32
port map (  data => source_post_odd_reg,
            clock => clk,
            clr => reset,
            enable => load_reg_post_odd,  -- helps to latch P3xorK3
or pre param odd
            q => post_param_odd);

-- handle the output
-- HAS TO UNDO THE LAST SWAP!!!
ciphertext_module: ciphertext
port map    (K4 => even_key,
             K5 => odd_key,
             K6 => even_plaintext,
             K7 => odd_plaintext,
             line0 => post_param_even,
             line1 => post_param_odd,
             line2 => pre_param_even,
             line3 => pre_param_odd,
             clk => clk,
             reset => reset,
             latch_ciphertext => latch_ciphertext,
             ciphertext => output);

-- This register holds the 128-bit key...
reg_key : keymodule
port map   (key => key,
            latch_key => latch_key,
            clk => clk,
            reset => reset,
            S0 => S0,
            S1 => S1,
            keyout => outkey);
end mixed;
```

8. EVENKEYGENERATOR

```
-------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja        ---
--- ID :  1669                                ---
--- Component : This component is the         ---
---             Evenkeygenerator              ---
--- Version : 1.0 - Beta                      ---
-------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

--Main: used to compute K-subkeys and as main buildig block of
enc/dec
entity evenkeygenerator is
port (input : in std_logic_vector(31 downto 0);
      in_S0 : in std_logic_vector(31 downto 0); --S0 for S-boxes in
h_fctn module
      in_S1 : in std_logic_vector(31 downto 0); --S1 for S-boxes in
h_fctn module
      clk : in std_logic;
      reset : in std_logic;
      outevenkey : out std_logic_vector(31 downto 0)); --output of
main on odd path
end evenkeygenerator;
```

```
--Building block of main
architecture mixed OF evenkeygenerator is
--
--component DECLARATION
--
--h-function modules
component h_fctn
port (input : in std_logic_vector(31 downto 0);
      S0 : in std_logic_vector(31 downto 0);
      S1 : in std_logic_vector(31 downto 0);
      output : out std_logic_vector(31 downto 0));
end component;

begin

--h-function transformation
h_block : h_fctn
port map (input => input,
          S0 => in_S0,
          S1 => in_S1,
       output => outevenkey);
end mixed;
```

9. EVENPLAINTEXTGENERATOR

```
-------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja          ---
--- ID :  1669                                  ---
--- Component : This component is the           ---
---             Evenplaintextgenerator          ---
--- Version : 1.0 - Beta                        ---
-------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

--Main: used to compute K-subkeys and as main buildig block of
enc/dec
entity evenplaintextgenerator is
port (input : in std_logic_vector(31 downto 0);
      in_S0 : in std_logic_vector(31 downto 0); --S0 for S-boxes in
h_fctn module
      in_S1 : in std_logic_vector(31 downto 0); --S1 for S-boxes in
h_fctn module
      clk : in std_logic;
      reset : in std_logic;
      outevenplaintext : out std_logic_vector(31 downto 0)); --output
of main on odd path
end evenplaintextgenerator;

--Building block of main
architecture mixed OF evenplaintextgenerator is
--
--component DECLARATION
--
--h-function modules
component h_fctn
port (input : in std_logic_vector(31 downto 0);
      S0 : in std_logic_vector(31 downto 0);
      S1 : in std_logic_vector(31 downto 0);
      output : out std_logic_vector(31 downto 0));
end component;

--Start Main's structure description
begin


--h-function transformation
h_block : h_fctn
port map (input => input,
          S0 => in_S0,
          S1 => in_S1,
          output => outevenplaintext);


end mixed;
```

10. H_FCTN

```
-------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja          ---
--- ID :  1669                                  ---
--- Component : This component is h-function.   ---
--- Version : 1.0 - Beta                        ---
-------------------------------------------------


-- -------- -----
-- input-->|S-box |---->|MDS|--> output
-- -------- -----

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

--h-function combines s-boxes and maximum distance separable matrix
entity h_fctn is
port( input : in std_logic_vector(31 downto 0);
      S0 : in std_logic_vector(31 downto 0);
      S1 : in std_logic_vector(31 downto 0);
      output : out std_logic_vector(31 downto 0));
end h_fctn;

--Building block of the h_fctn
architecture struct OF h_fctn is

component S_boxes
port( income : in std_logic_vector(31 downto 0);
      S0 : in std_logic_vector(31 downto 0);
      S1 : in std_logic_vector(31 downto 0);
      outcome : out std_logic_vector(31 downto 0));
end component;
--maximum distance separable matrix
component mds
port( y0 : in std_logic_vector(7 downto 0);
      y1 : in std_logic_vector(7 downto 0);
      y2 : in std_logic_vector(7 downto 0);
      y3 : in std_logic_vector(7 downto 0);
      z0 : out std_logic_vector(7 downto 0);
```

```
      z1 : out std_logic_vector(7 downto 0);
      z2 : out std_logic_vector(7 downto 0);
      z3 : out std_logic_vector(7 downto 0));
end component;
Signal
out_mds, --output of the MDS
out_S_box : std_logic_vector(31 downto 0); --output of the S-boxes
--Start of the h-function structure
BEGin
--S-box that permutes the input and XORs it with S0 and S1
S_box1 : S_boxes
port map( income => input,
          S0 => S0,
          S1 => S1,
          outcome => out_S_box);

--MDS
mds1 : mds
port map( y0 => out_S_box(7 downto 0),
          y1 => out_S_box(15 downto 8),
          y2 => out_S_box(23 downto 16),
          y3 => out_S_box(31 downto 24),
          z0 => out_mds(7 downto 0),
          z1 => out_mds(15 downto 8),
          z2 => out_mds(23 downto 16),
          z3 => out_mds(31 downto 24));
--Assigning output its value form signal
output <= out_mds;
end struct;
```

11. KEYMODULE

```
-------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja              ---
--- ID :  1669                                      ---
--- Component : This component is the keymodule     ---
--- Version : 1.0 - Beta                            ---
-------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

--              128-bit key
--                  |
-- latch_key ---> keyreg
-- |                | |
-- |-----------> ---------
-- |              |    |
-- |             RS_matrix
-- |    ------------|-----------
-- |    |                       |
-- --> S0reg                    |
--      |                       |
--      S0                      S1

entity keymodule is
port (key : in std_logic_vector(127 downto 0);
      latch_key : in std_logic;
      clk : in std_logic;
      reset : in std_logic;
      S0,S1 : out std_logic_vector(31 downto 0);
      keyout : out std_logic_vector(127 downto 0));
end keymodule;

architecture mixed of keymodule is

component rsmatrix
port (m0,m1,m2,m3,m4,m5,m6,m7 : in std_logic_vector(7 downto 0);
      s0,s1,s2,s3 : out std_logic_vector(7 downto 0));
end component;

component reg128
port(clock,clr,enable : in  std_logic;
     data : in std_logic_vector (127 downto 0);
     q : out std_logic_vector (127 downto 0));
end component;

component reg32
port(clock,clr,enable : in  std_logic;
     data : in std_logic_vector (31 downto 0);
     q : out std_logic_vector (31 downto 0));
end component;

component mux_2x64
port (sel : in std_logic;
      in_0 : in std_logic_vector(63 downto 0);
      in_1 : in std_logic_vector(63 downto 0);
      output : out std_logic_vector(63 downto 0));
end component;

signal RS_input : std_logic_vector(63 downto 0);
signal RS_output : std_logic_vector(31 downto 0);
signal regkey_output : std_logic_vector(127 downto 0);

begin
keyreg : reg128
port map(data => key,
         clock => clk,
         clr => reset,
         enable => latch_key,
         q => regkey_output);

         keyout <= regkey_output;

-- choose input of RS_Matrix
RSinputChoice: mux_2x64
port map(sel => latch_key,
         in_0 => regkey_output(127 downto 64),
         in_1 => key(63 downto 0),
         output => RS_input);

-- compute S
compute_S: rsmatrix
port map (m0 => RS_input(7 downto 0),
          m1 => RS_input(15 downto 8),
          m2 => RS_input(23 downto 16),
          m3 => RS_input(31 downto 24),
          m4 => RS_input(39 downto 32),
          m5 => RS_input(47 downto 40),
          m6 => RS_input(55 downto 48),
```

```
          m7 => RS_input(63 downto 56),
          s0 => RS_output(7 downto 0),
          s1 => RS_output(15 downto 8),
          s2 => RS_output(23 downto 16),
           s3 => RS_output(31 downto 24));
-- store S0 if it's being computed
S0reg : reg32
      port map(data => RS_output,
               clock => clk,
               clr => reset,
               enable => latch_key,
               q => S0);
-- latch S1 to the output... Will be correct
-- soon after latch_key drops to 0.
S1 <= RS_output;
end mixed;
```

12. LITTLE_ENDIAN_CONVERTER

```
-------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja          ---
--- ID :  1669                                  ---
--- Component : This component is the little    ---
---              converter.                     ---
--- Version : 1.0 - Beta                        ---
-------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity little_endian_converter is
port (big_endian : in std_logic_vector(127 downto 0);
      little_endian : out std_logic_vector(127 downto 0));
end little_endian_converter;

architecture behavior of little_endian_converter is

signal
le0,le1,le2,le3,le4,le5,le6,le7,le8,le9,le10,le11,le12,le13,le14,le15
       : std_logic_vector(7 downto 0);

begin
      process(big_endian)
      begin
      --little endian convertion
      le0 <=big_endian(7 downto 0);
      le1 <=big_endian(15 downto 8);
      le2 <=big_endian(23 downto 16);
      le3 <=big_endian(31 downto 24);
      le4 <=big_endian(39 downto 32);
      le5 <=big_endian(47 downto 40);
      le6 <=big_endian(55 downto 48);
      le7 <=big_endian(63 downto 56);
      le8 <=big_endian(71 downto 64);
      le9 <=big_endian(79 downto 72);
      le10 <=big_endian(87 downto 80);
      le11 <=big_endian(95 downto 88);
      le12 <=big_endian(103 downto 96);
      le13 <=big_endian(111 downto 104);
      le14 <=big_endian(119 downto 112);
      le15 <=big_endian(127 downto 120);

      end process;

little_endian <= le0 & le1 & le2 & le3 & le4 & le5 & le6 & le7 & le8
& le9 & le10 & le11 & le12 & le13 & le14 & le15;

end behavior;
```

13. MDS

```
-------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja          ---
--- ID :  1669                                  ---
--- Component : This component is the MDS.      ---
--- Version : 1.0 - Beta                        ---
-------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;

-- Maximum Distance Separable matrix
--
-- Each entry is 8-bits wide. The following operation is
-- performed in the Galois field GF(2^8).
--
-- |z0|           |01 EF 5B 5B|           |y0|
-- |z1| =         |5B EF EF 01|   o       |y1|
-- |z2|           |EF 5B 01 EF|           |y2|
-- |z3|           |EF 01 EF 5B|           |y3|
--
entity mds is
port (y0 : in std_logic_vector(7 downto 0);
      y1 : in std_logic_vector(7 downto 0);
      y2 : in std_logic_vector(7 downto 0);
      y3 : in std_logic_vector(7 downto 0);
      z0 : out std_logic_vector(7 downto 0);
      z1 : out std_logic_vector(7 downto 0);
      z2 : out std_logic_vector(7 downto 0);
      z3 : out std_logic_vector(7 downto 0));
end mds;

architecture struct of mds is
-- multiplication by 5B
component multgf_5B
port (x : in std_logic_vector(7 downto 0);
      y : out std_logic_vector(7 downto 0));
end component;

-- multiplication by EF
component multgf_EF
port (x : in std_logic_vector(7 downto 0);
      y : out std_logic_vector(7 downto 0));
end component;

-- bit-wise xor of 4 8-bit vectors
component xor_4x8
```

```
port (x : in std_logic_vector (7 DOWNTO 0);
      y : in std_logic_vector (7 DOWNTO 0);
      z : in std_logic_vector (7 DOWNTO 0);
      w : in std_logic_vector (7 DOWNTO 0);
      result : out std_logic_vector (7 DOWNTO 0));
end component;

-- result of multiplications
signal y0xEF : std_logic_vector(7 downto 0);
signal y0x5B : std_logic_vector(7 downto 0);
signal y1xEF : std_logic_vector(7 downto 0);
signal y1x5B : std_logic_vector(7 downto 0);
signal y2xEF : std_logic_vector(7 downto 0);
signal y2x5B : std_logic_vector(7 downto 0);
signal y3xEF : std_logic_vector(7 downto 0);
signal y3x5B : std_logic_vector(7 downto 0);
begin
--
-- Do multiplications
--
y0xEF_comp: multgf_EF
port map ( x => y0,y => y0xEF);
y0x5B_comp: multgf_5B
port map ( x => y0,y => y0x5B);
y1xEF_comp: multgf_EF
port map ( x => y1,y => y1xEF);
y1x5B_comp: multgf_5B
port map ( x => y1,y => y1x5B);
y2xEF_comp: multgf_EF
port map ( x => y2,y => y2xEF);
y2x5B_comp: multgf_5B
port map ( x => y2,y => y2x5B);
y3xEF_comp: multgf_EF
port map ( x => y3,y => y3xEF);
y3x5B_comp: multgf_5B
port map ( x => y3,y => y3x5B);

--
-- do the additions (row by row)
--
-- (An addition in GF corresponds to an XOR.)
--
-- z0 = y0x01 + y1xEF + y2x5B + y3x5B
z0_xor: xor_4x8
port map (x => y0,
          y => y1xEF,
          z => y2x5B,
          w => y3x5B,
          result => z0 );
-- z1 = y0x5B + y1xEF + y2xEF + y3x01
z1_xor: xor_4x8
port map ( x => y0x5B,
           y => y1xEF,
           z => y2xEF,
           w => y3,
           result => z1 );
-- z2 = y0xEF + y1x5B + y2x01 + y3xEF
z2_xor: xor_4x8
port map ( x => y0xEF,
           y => y1x5B,
           z => y2,
           w => y3xEF,
           result => z2 );
-- z3 = y0xEF + y1x01 + y2xEF + y3x5B
z3_xor: xor_4x8
port map  ( x => y0xEF,
            y => y1,
            z => y2xEF,
            w => y3x5B,
            result => z3 );

end struct;
library ieee;
use ieee.std_logic_1164.all;

-- Multiplication of an 8-bit vector by 0h5B in
-- Galois field GF9(2^8).
--
-- x: input vector
-- y: output vector
--
entity multgf_5B is
port(
x : in std_logic_vector(7 downto 0);
y : out std_logic_vector(7 downto 0));
end multgf_5B;

architecture struct of multgf_5B is
begin
y(0) <= x(2) XOR x(0);
y(1) <= x(3) XOR x(1) XOR x(0);
y(2) <= x(4) XOR x(2) XOR x(1);
y(3) <= x(5) XOR x(3) XOR x(0);
y(4) <= x(6) XOR x(4) XOR x(1) XOR x(0);
y(5) <= x(7) XOR x(5) XOR x(1);
y(6) <= x(6) XOR x(0);
y(7) <= x(7) XOR x(1);
end struct;

library ieee;
use ieee.std_logic_1164.all;

-- Multiplication of an 8-bit vector by 0hEF in
-- Galois field GF9(2^8). See section 4.2 of
-- "Twofish: A 128-Bit Block Cipher" for
-- details.
--
-- x: input vector
-- y: output vector
--
entity multgf_EF is
port(
x : in std_logic_vector(7 downto 0);
y : out std_logic_vector(7 downto 0));
end multgf_EF;

architecture struct of multgf_EF is
begin
y(0) <= x(2) XOR x(1) XOR x(0);
y(1) <= x(3) XOR x(2) XOR x(1) XOR x(0);
y(2) <= x(4) XOR x(3) XOR x(2) XOR x(1) XOR x(0);
y(3) <= x(5) XOR x(4) XOR x(3) XOR x(0);
```

```
y(4) <= x(6) XOR x(5) XOR x(4) XOR x(1);
y(5) <= x(7) XOR x(6) XOR x(5) XOR x(1) XOR x(0);
y(6) <= x(7) XOR x(6) XOR x(0);
y(7) <= x(7) XOR x(1) XOR x(0);
end struct;
```

14. MUX_2X32

```
------------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                         ---
--- ID :  1669                                                 ---
--- Component : This component is a 2:1 multiplexor. (32bit)   ---
--- Version : 1.0 - Beta                                       ---
------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
-- Multiplexer that selects between in_0 and in_1.
-- in_0 is returned if sel is 0, in_1 is returned
-- otherwise.

entity mux_2x32 is
port (sel : in std_logic;
      in_0 : in std_logic_vector(31 downto 0);
      in_1 : in std_logic_vector(31 downto 0);
      output : out std_logic_vector(31 downto 0));
end mux_2x32;

architecture behavior of mux_2x32 is
begin
mux : process(sel,in_0,in_1)
begin
if sel = '0' then
      output <= in_0;
else
      output <= in_1;
end if;
end process;
end behavior;
```

15. MUX_2X64

```
------------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                         ---
--- ID :  1669                                                 ---
--- Component : This component is a 2:1 multiplexor. (64bit)   ---
--- Version : 1.0 - Beta                                       ---
------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
-- Multiplexer that selects between in_0 and in_1.
-- in_0 is returned if sel is 0, in_1 is returned
-- otherwise.

entity mux_2x64 is
port (sel : in std_logic;
      in_0 : in std_logic_vector(63 downto 0);
      in_1 : in std_logic_vector(63 downto 0);
      output : out std_logic_vector(63 downto 0));
end mux_2x64;

architecture behavior of mux_2x64 is
begin
mux : process(sel,in_0,in_1)
begin
if sel = '0' then
      output <= in_0;
else
      output <= in_1;
end if;
end process;
end behavior;
```

16. ODDKEYGENERATOR

```
------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja             ---
--- ID :  1669                                     ---
--- Component : This component is the              ---
---             Oddkeygenerator                    ---
--- Version : 1.0 - Beta                           ---
------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

--Main: used to compute K-subkeys and as main buildig block of
enc/dec
entity oddkeygenerator is
port (input : in std_logic_vector(31 downto 0);
      in_S0 : in std_logic_vector(31 downto 0); --S0 for S-boxes in
h_fctn module
      in_S1 : in std_logic_vector(31 downto 0); --S1 for S-boxes in
h_fctn module
      clk : in std_logic;
      reset : in std_logic;
      outoddkey : out std_logic_vector(31 downto 0)); --output of
main odd key
end oddkeygenerator;

--Building block of main
architecture mixed OF oddkeygenerator is
--
--component DECLARATION
--
--h-function modules
component h_fctn
port (input : in std_logic_vector(31 downto 0);
      S0 : in std_logic_vector(31 downto 0);
      S1 : in std_logic_vector(31 downto 0);
      output : out std_logic_vector(31 downto 0));
end component;

--rotate left 8 bit
component rol8_32
port (data : in std_logic_vector(31 downto 0);
      q : out std_logic_vector(31 downto 0));
end component;
```

```
--SIGNAL DECLARATION
--

Signal sig_out_h : std_logic_vector(31 downto 0); --output of
h_function

--Start Main's structure description
begin

--h-function transformation
h_block : h_fctn
port map (input => input,
          S0 => in_S0,
          S1 => in_S1,
          output => sig_out_h);

--rotate to the left the output of the h_function 8 times
rol8_sig_out_h: rol8_32
port map (data => sig_out_h,
          q => outoddkey);

end mixed;
```

17. ODDPLAINTEXTGENERATOR

```
-------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja          ---
--- ID :  1669                                  ---
--- Component : This component is the           ---
---             Oddplaintextgenerator           ---
--- Version : 1.0 - Beta                        ---
-------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

--Main: used to compute K-subkeys and as main buildig block of
enc/dec
entity oddplaintextgenerator is
port (input : in std_logic_vector(31 downto 0);
      in_S0 : in std_logic_vector(31 downto 0); --S0 for S-boxes in
h_fctn module
      in_S1 : in std_logic_vector(31 downto 0); --S1 for S-boxes in
h_fctn module
      clk : in std_logic;
      reset : in std_logic;
      sel_odd : in std_logic; --input selecting btw even path or odd
path
      sel_k : in std_logic; --input selecting btw calculating a key
or an encryption/decryption
      outoddplaintext : out std_logic_vector(31 downto 0)); --output
of main on odd path
end oddplaintextgenerator;

--Building block of main
architecture mixed OF oddplaintextgenerator is
--
--component DECLARATION
--
--h-function modules
component h_fctn
port (input : in std_logic_vector(31 downto 0);
      S0 : in std_logic_vector(31 downto 0);
      S1 : in std_logic_vector(31 downto 0);
      output : out std_logic_vector(31 downto 0));
end component;

--Multiplexer modules
component mux_2x32
port(sel : in std_logic;
      in_0 : in std_logic_vector(31 downto 0);
      in_1 : in std_logic_vector(31 downto 0);
      output : out std_logic_vector(31 downto 0));
end component;

--register
component reg32
port(clock,clr,enable : in  std_logic;
     data : in std_logic_vector (31 downto 0);
     q : out std_logic_vector (31 downto 0));
end component;

--rotate left 8 bit
component rol8_32
port (data : in std_logic_vector(31 downto 0);
      q : out std_logic_vector(31 downto 0));
end component;

--
--SIGNAL DECLARATION
--
Signal
      sel_odd_k,
      sel_odd_result : std_logic;
Signal
      rotated_input, --P1 Xor K1 rotated left 8 times
      sig_in_h, --input to h_function
      sig_out_h, --output of h_function
      rotated_out_h --output of the h_fctn rol 8 times
   : std_logic_vector(31 downto 0); --output of the main on the odd
path

--Start Main's structure description
begin
-- control
sel_odd_k <= sel_odd AND sel_k;
sel_odd_result <= sel_odd AND (NOT sel_k);

--Rotate input to the left 8 time
rol8_input: rol8_32
port map (data => input,
          q => rotated_input);

-- Choose input of h-function: rotate only if we
-- we are compute an odd result (and not a k!)
mux_choose_hin : mux_2x32
port map(sel => sel_odd_result,       -- sel_odd_results = sel_odd &&
(not sel_k)
```

```
          in_0 => input,                          -- for encryption,
sel_k will be 0, so if sel_odd is equal to 0, sel will choose even
path else if sel_odd equals to 1 , sel will choose odd path.
          in_1 => rotated_input,
          output => sig_in_h);

-------------------------------------------------------------
--    Sel_odd_results --      Sel_odd     &&    NOTSel_k    --
-------------------------------------------------------------
--          0 <==             0     --          0
--
--          0 <==             0     --          1
--
--          1 <==             1     --          0
--
--          0 <==             1     --          1
--
-------------------------------------------------------------

--h-function transformation
h_block : h_fctn
port map (input => sig_in_h,
          S0 => in_S0,
          S1 => in_S1,
          output => sig_out_h);

--rotate to the left the output of the h_function 8 times
rol8_sig_out_h: rol8_32
port map (data => sig_out_h,
          q => rotated_out_h);

--Multiplex between out_h and rotated out_h
mux_out_h_rotated_out_h : mux_2x32
port map(sel => sel_odd_k,          -- sel_odd_k = sel_odd && sel_k
          in_1 => rotated_out_h,          -- rotated_out_h will only
be selected if it is on the path and the current process is key
scheduling
          in_0 => sig_out_h,              -- All other condition,
output would be sig_out_h
          output => outoddplaintext);

      ---------------------------------------------------
      --    Sel_odd_k   --    Sel_odd     &&    Sel_k     --
      ---------------------------------------------------
      --    0           <==   0     --          0         --
      --    0           <==   0     --          1         --
      --    0           <==   1     --          0         --
      --    1           <==   1     --          1         --
      ---------------------------------------------------

end mixed;
```

18. OPSELECT

```
-------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                    ---
--- ID :  1669                                            ---
--- Component : This component is the operation selector. ---
--- Version : 1.0 - Beta                                  ---
-------------------------------------------------------------

-- This module performs the operation
-- that defines a block as a decrypter
-- or encrypter.

library ieee;
use ieee.std_logic_1164.all;

-- Operation selector
--
-- This module includes all parts of the
-- cipher that differ if it is in encryption
-- or decryption mode.
--
-- Encryption:
--                  in_A     in_B
--                   |        |
--                   |        |
--  F-fct_A ----> xor      <<< 1
--  F-function     |        |
--  F-fct_B ---------------> xor
--                   |        |
--                 >>> 1      |
--                   |        |
--                 out_A    out_B
--
--
-- Decryption:
--                  in_A     in_B
--                   |        |
--                 <<< 1      |
--                   |        |
--  F-fct_A ----> xor         |
--  F-function     |          |
--  F-fct_B ---------------> xor
--                   |        |
--                   |      >>> 1
--                   |        |
--                 out_A    out_B
--
--
-- encrypt: encryption mode / not decryption mode
-- F-fct_A: even output of F-function (top)
-- F-fct_B: odd output of F-function (bottom)
-- in_A: third input to the round
-- in_B: fourth input to the round
-- out_A: third output of the round
-- out_B: fourth output of the round
---

entity opselect is
port (encrypt : in std_logic;
      F_fct_A : in std_logic_vector(31 downto 0);
      F_fct_B : in std_logic_vector(31 downto 0);
      in_A : in std_logic_vector(31 downto 0);
      in_B : in std_logic_vector(31 downto 0);
      out_A : out std_logic_vector(31 downto 0);
      out_B : out std_logic_vector(31 downto 0));
end opselect;

architecture struct of opselect is
```

```
-- subcomponent
component sub_opselect
port (rotate_before : in std_logic;
      F_fct : in std_logic_vector(31 downto 0);
      input : in std_logic_vector(31 downto 0);
      output : out std_logic_vector(31 downto 0));
end component;
signal decrypt : std_logic;
begin
-- left subcomponent:
-- do: rol, xor for decryption
-- xor, ror for encryption
-- => "rotate_before" when "encrypt" is 0
leftsub : sub_opselect
port map (rotate_before => decrypt,
            F_fct => F_fct_A,
            input => in_A,
            output => out_A);
-- right subcomponent:
-- do: rol, xor for encryption
-- xor, ror for decryption
-- => "rotate_before" when "encrypt" is 1
rightsub : sub_opselect
port map (rotate_before => encrypt,
            F_fct => F_fct_B,
            input => in_B,
            output => out_B);


      decrypt <= not encrypt;
end struct;
```

19. PERM_Q0

```
-------------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                          ---
--- ID :  1669                                                  ---
--- Component : This component is the q0_permutation.           ---
--- Version : 1.0 - Beta                                        ---
-------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

--Perm_q0: Block used for q0-type of permutation
entity Perm_q0 is
port( input : in std_logic_vector(7 downto 0);
      output : out std_logic_vector(7 downto 0));
end Perm_q0;

architecture struct of Perm_q0 is
component xor_2x4
port(a : in std_logic_vector(3 downto 0);
     b : in std_logic_vector(3 downto 0);
     result : out std_logic_vector(3 downto 0));
end component;

component xor_3x4
port(dataa : in std_logic_vector(3 downto 0);
     datab : in std_logic_vector(3 downto 0);
     datac : in std_logic_vector(3 downto 0);
     result : out std_logic_vector(3 downto 0));
end component;

signal
a_xor_b, --a XOR b
rotated_b, --b rotated right one time
a_zero, --(a(0),0,0,0) vector
befor_t1, --signal before lookup table t1
a_prime, --a prime (output of lookup table t0)
b_prime, --b prime (output of lookup table t1)
ap_xor_bp, --a prime XOR b prime
rotated_bp, --b prime rotated right one time
ap_zero, --(a'(0),0,0,0) vector
befor_t3 : std_logic_vector(3 downto 0); --input signal to lookup
table t3

signal
permuted_input : std_logic_vector(7 downto 0); --output of the q0
block

--Start of q0 permutation structure
begin
-- rotate b right once
-- b(3 downto 0) is input(3 downto 0)
rotated_b(3) <= input(0);
rotated_b(2) <= input(3);
rotated_b(1) <= input(2);
rotated_b(0) <= input(1);
--build a vector with (a(0),0,0,0)
-- a(3 downto 0) is input(7 downto 4)
a_zero(3) <= input(4);
a_zero(2) <= '0';
a_zero(1) <= '0';
a_zero(0) <= '0';
--rotate b_prime right once
rotated_bp(3) <= b_prime(0);
rotated_bp(2) <= b_prime(3);
rotated_bp(1) <= b_prime(2);
rotated_bp(0) <= b_prime(1);
--build a vector with (ap(0),0,0,0)
ap_zero(3) <= a_prime(0);
ap_zero(2) <= '0';
ap_zero(1) <= '0';
ap_zero(0) <= '0';
--
-- Compute a xor b
--
--xor a and b
xor1 : xor_2x4
port map( a => input(7 downto 4),
          b => input(3 downto 0),
          result => a_xor_b);
--
-- Compute a xor (b >>> 1) xor (a(0),0,0,0)
--
--xor a, rotated b and a_zero vector
xor2 : xor_3x4
port map( dataa => input(7 downto 4),
          datab => rotated_b,
```

```
         datac => a_zero,
         result => befor_t1);
--
-- Compute a' xor b'
--
--xor a_prime and b_prime
xor3 : xor_2x4
port map( a => a_prime,
         b => b_prime,
         result => ap_xor_bp);
--
-- a' xor (b' >>> 1) xor (a'(0),0,0,0)
--
--xor rotated b_prime, a_prime and ap_zero vector
xor4 : xor_3x4
port map( dataa => a_prime,
         datab => rotated_bp,
         datac => ap_zero,
         result => befor_t3);
--
--
-- lookup table t0
--
q0_t0 : process (a_xor_b, a_prime)
begin
if a_xor_b = "0000" then a_prime <= "1000"; -- 8
elsif a_xor_b = "0001" then a_prime <= "0001"; -- 1
elsif a_xor_b = "0010" then a_prime <= "0111"; -- 7
elsif a_xor_b = "0011" then a_prime <= "1101"; -- D
elsif a_xor_b = "0100" then a_prime <= "0110"; -- 6
elsif a_xor_b = "0101" then a_prime <= "1111"; -- F
elsif a_xor_b = "0110" then a_prime <= "0011"; -- 3
elsif a_xor_b = "0111" then a_prime <= "0010"; -- 2
elsif a_xor_b = "1000" then a_prime <= "0000"; -- 0
elsif a_xor_b = "1001" then a_prime <= "1011"; -- B
elsif a_xor_b = "1010" then a_prime <= "0101"; -- 5
elsif a_xor_b = "1011" then a_prime <= "1001"; -- 9
elsif a_xor_b = "1100" then a_prime <= "1110"; -- E
elsif a_xor_b = "1101" then a_prime <= "1100"; -- C
elsif a_xor_b = "1110" then a_prime <= "1010"; -- A
else a_prime <= "0100"; -- 4
end if;
end process;
--
-- lookup table t1
--
q0_t1 : process (befor_t1, b_prime)
begin
if befor_t1 = "0000" then b_prime <= "1110"; -- E
elsif befor_t1 = "0001" then b_prime <= "1100"; -- C
elsif befor_t1 = "0010" then b_prime <= "1011"; -- B
elsif befor_t1 = "0011" then b_prime <= "1000"; -- 8
elsif befor_t1 = "0100" then b_prime <= "0001"; -- 1
elsif befor_t1 = "0101" then b_prime <= "0010"; -- 2
elsif befor_t1 = "0110" then b_prime <= "0011"; -- 3
elsif befor_t1 = "0111" then b_prime <= "0101"; -- 5
elsif befor_t1 = "1000" then b_prime <= "1111"; -- F
elsif befor_t1 = "1001" then b_prime <= "0100"; -- 4
elsif befor_t1 = "1010" then b_prime <= "1010"; -- A
elsif befor_t1 = "1011" then b_prime <= "0110"; -- 6
elsif befor_t1 = "1100" then b_prime <= "0111"; -- 7
elsif befor_t1 = "1101" then b_prime <= "0000"; -- 0
elsif befor_t1 = "1110" then b_prime <= "1001"; -- 9
else b_prime <= "1101"; -- D
end if;
end process;
--
-- lookup table t2
--
q0_t2 : process (ap_xor_bp,permuted_input)
begin
if ap_xor_bp = "0000" then permuted_input(7 downto 4) <= "1011"; -- B
elsif ap_xor_bp = "0001" then permuted_input(7 downto 4) <= "1010"; -
- A
elsif ap_xor_bp = "0010" then permuted_input(7 downto 4) <= "0101"; -
- 5
elsif ap_xor_bp = "0011" then permuted_input(7 downto 4) <= "1110"; -
- E
elsif ap_xor_bp = "0100" then permuted_input(7 downto 4) <= "0110"; -
- 6
elsif ap_xor_bp = "0101" then permuted_input(7 downto 4) <= "1101"; -
- D
elsif ap_xor_bp = "0110" then permuted_input(7 downto 4) <= "1001"; -
- 9
elsif ap_xor_bp = "0111" then permuted_input(7 downto 4) <= "0000"; -
- 0
elsif ap_xor_bp = "1000" then permuted_input(7 downto 4) <= "1100"; -
- C
elsif ap_xor_bp = "1001" then permuted_input(7 downto 4) <= "1000"; -
- 8
elsif ap_xor_bp = "1010" then permuted_input(7 downto 4) <= "1111"; -
- F
elsif ap_xor_bp = "1011" then permuted_input(7 downto 4) <= "0011"; -
- 3
elsif ap_xor_bp = "1100" then permuted_input(7 downto 4) <= "0010"; -
- 2
elsif ap_xor_bp = "1101" then permuted_input(7 downto 4) <= "0100"; -
- 4
elsif ap_xor_bp = "1110" then permuted_input(7 downto 4) <= "0111"; -
- 7
else permuted_input(7 downto 4) <= "0001"; -- 1
end if;
end process q0_t2;
--lookup table t3
q0_t3 : process (befor_t3,permuted_input)
begin
if befor_t3 = "0000" then permuted_input(3 downto 0) <= "1101"; -- D
elsif befor_t3 = "0001" then permuted_input(3 downto 0) <= "0111"; --
7
elsif befor_t3 = "0010" then permuted_input(3 downto 0) <= "1111"; --
F
elsif befor_t3 = "0011" then permuted_input(3 downto 0) <= "0100"; --
4
elsif befor_t3 = "0100" then permuted_input(3 downto 0) <= "0001"; --
1
elsif befor_t3 = "0101" then permuted_input(3 downto 0) <= "0010"; --
2
elsif befor_t3 = "0110" then permuted_input(3 downto 0) <= "0110"; --
6
elsif befor_t3 = "0111" then permuted_input(3 downto 0) <= "1110"; --
E
```

```
elsif befor_t3 = "1000" then permuted_input(3 downto 0) <= "1001"; --
9
elsif befor_t3 = "1001" then permuted_input(3 downto 0) <= "1011"; --
B
elsif befor_t3 = "1010" then permuted_input(3 downto 0) <= "0011"; --
3
elsif befor_t3 = "1011" then permuted_input(3 downto 0) <= "0000"; --
0
elsif befor_t3 = "1100" then permuted_input(3 downto 0) <= "1000"; --
8
elsif befor_t3 = "1101" then permuted_input(3 downto 0) <= "0101"; --
5
elsif befor_t3 = "1110" then permuted_input(3 downto 0) <= "1100"; --
C
else permuted_input(3 downto 0) <= "1010"; -- A
end if;
end process ;
--assigning value to the output from signal
-- output = b&a (see page 10 of twofish)
output <= permuted_input(3 downto 0) & permuted_input(7 downto 4);
end struct;
```

20. PERM_Q1

```
-------------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                          ---
--- ID :  1669                                                  ---
--- Component : This component is the q1_permutation.           ---
--- Version : 1.0 - Beta                                        ---
-------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

--Perm_q1: Block used for q1-type of permutation
entity Perm_q1 is
port( input : in std_logic_vector(7 downto 0);
          output : out std_logic_vector(7 downto 0));
end Perm_q1;

architecture struct OF Perm_q1 IS

component xor_2x4
port(a : in std_logic_vector(3 downto 0);
     b : in std_logic_vector(3 downto 0);
     result : out std_logic_vector(3 downto 0));
end component;

component xor_3x4
port(dataa : in std_logic_vector(3 downto 0);
           datab : in std_logic_vector(3 downto 0);
           datac : in std_logic_vector(3 downto 0);
           result : out std_logic_vector(3 downto 0));
end component;

Signal
a_xor_b, --a XOR b
rotated_b, --b rotated right one time
a_zero, --(a(0),0,0,0) vector
befor_t1, --signal before lookup table t1
a_prime, --a prime (output of lookup table t0)
b_prime, --b prime (output of lookup table t1)
ap_xor_bp, --a prime XOR b prime
rotated_bp, --b prime rotated right one time
ap_zero, --(a'(0),0,0,0) vector
befor_t3 : std_logic_vector(3 downto 0); --input signal to lookup
table t3

Signal
permuted_input : std_logic_vector(7 downto 0); --output of the q0
block

--Start of q0 permutation structure
begin
-- input = a&b
-- rotate b right once
-- b(3 downto 0) is input(3 downto 0)
rotated_b(3) <= input(0);
rotated_b(2) <= input(3);
rotated_b(1) <= input(2);
rotated_b(0) <= input(1);
--build a vector with (a(0),0,0,0)
-- a(3 downto 0) is input(7 downto 4)
a_zero(3) <= input(4);
a_zero(2) <= '0';
a_zero(1) <= '0';
a_zero(0) <= '0';
--rotate b_prime right once
rotated_bp(3) <= b_prime(0);
rotated_bp(2) <= b_prime(3);
rotated_bp(1) <= b_prime(2);
rotated_bp(0) <= b_prime(1);
--build a vector with (ap(0),0,0,0)
ap_zero(3) <= a_prime(0);
ap_zero(2) <= '0';
ap_zero(1) <= '0';
ap_zero(0) <= '0';
--
-- Computer a xor b
--

--xor a and b
xor1 : xor_2x4
port MAP( a => input(7 downto 4),
          b => input(3 downto 0),
          result => a_xor_b);
--
-- Compute a xor (b >>> 1) xor (a(0),0,0,0)
--
--xor a, rotated b and a_zero vector
xor2 : xor_3x4
port MAP( dataa => input(7 downto 4),
          datab => rotated_b,
          datac => a_zero,
          result => befor_t1);

--xor a_prime and b_prime
xor3 : xor_2x4
port MAP( a => a_prime,
          b => b_prime,
```

```
        result => ap_xor_bp);

--
-- a' xor (b' >>> 1) xor (a'(0),0,0,0)
--
--xor rotated b_prime, a_prime and ap_zero vector
xor4 : xor_3x4
port MAP( dataa => a_prime,
          datab => rotated_bp,
          datac => ap_zero,
          result => befor_t3);
--
-- lookup table t0
--
q1_t0 : process (a_xor_b,a_prime)
begin
if a_xor_b = "0000" then a_prime <= "0010"; -- 2
elsif a_xor_b = "0001" then a_prime <= "1000"; -- 8
elsif a_xor_b = "0010" then a_prime <= "1011"; -- B
elsif a_xor_b = "0011" then a_prime <= "1101"; -- D
elsif a_xor_b = "0100" then a_prime <= "1111"; -- F
elsif a_xor_b = "0101" then a_prime <= "0111"; -- 7
elsif a_xor_b = "0110" then a_prime <= "0110"; -- 6
elsif a_xor_b = "0111" then a_prime <= "1110"; -- E
elsif a_xor_b = "1000" then a_prime <= "0011"; -- 3
elsif a_xor_b = "1001" then a_prime <= "0001"; -- 1
elsif a_xor_b = "1010" then a_prime <= "1001"; -- 9
elsif a_xor_b = "1011" then a_prime <= "0100"; -- 4
elsif a_xor_b = "1100" then a_prime <= "0000"; -- 0
elsif a_xor_b = "1101" then a_prime <= "1010"; -- A
elsif a_xor_b = "1110" then a_prime <= "1100"; -- C
else a_prime <= "0101"; -- 5
end if;
end process;
--lookup table t1
q1_t1 : process (befor_t1,b_prime)
begin
if befor_t1 = "0000" then b_prime <= "0001"; -- 1
elsif befor_t1 = "0001" then b_prime <= "1110"; -- E
elsif befor_t1 = "0010" then b_prime <= "0010"; -- 2
elsif befor_t1 = "0011" then b_prime <= "1011"; -- B
elsif befor_t1 = "0100" then b_prime <= "0100"; -- 4
elsif befor_t1 = "0101" then b_prime <= "1100"; -- C
elsif befor_t1 = "0110" then b_prime <= "0011"; -- 3
elsif befor_t1 = "0111" then b_prime <= "0111"; -- 7
elsif befor_t1 = "1000" then b_prime <= "0110"; -- 6
elsif befor_t1 = "1001" then b_prime <= "1101"; -- D
elsif befor_t1 = "1010" then b_prime <= "1010"; -- A
elsif befor_t1 = "1011" then b_prime <= "0101"; -- 5
elsif befor_t1 = "1100" then b_prime <= "1111"; -- F
elsif befor_t1 = "1101" then b_prime <= "1001"; -- 9
elsif befor_t1 = "1110" then b_prime <= "0000"; -- 0
else b_prime <= "1000"; -- 8
end if;
end process;
--lookup table t2
q1_t2 : process (ap_xor_bp,permuted_input)
begin
if ap_xor_bp = "0000" then permuted_input(7 downto 4) <= "0100"; -- 4
elsif ap_xor_bp = "0001" then permuted_input(7 downto 4) <= "1100"; -
- C
elsif ap_xor_bp = "0010" then permuted_input(7 downto 4) <= "0111"; -
- 7
elsif ap_xor_bp = "0011" then permuted_input(7 downto 4) <= "0101"; -
- 5
elsif ap_xor_bp = "0100" then permuted_input(7 downto 4) <= "0001"; -
- 1
elsif ap_xor_bp = "0101" then permuted_input(7 downto 4) <= "0110"; -
- 6
elsif ap_xor_bp = "0110" then permuted_input(7 downto 4) <= "1001"; -
- 9
elsif ap_xor_bp = "0111" then permuted_input(7 downto 4) <= "1010"; -
- A
elsif ap_xor_bp = "1000" then permuted_input(7 downto 4) <= "0000"; -
- 0
elsif ap_xor_bp = "1001" then permuted_input(7 downto 4) <= "1110"; -
- E
elsif ap_xor_bp = "1010" then permuted_input(7 downto 4) <= "1101"; -
- D
elsif ap_xor_bp = "1011" then permuted_input(7 downto 4) <= "1000"; -
- 8
elsif ap_xor_bp = "1100" then permuted_input(7 downto 4) <= "0010"; -
- 2
elsif ap_xor_bp = "1101" then permuted_input(7 downto 4) <= "1011"; -
- B
elsif ap_xor_bp = "1110" then permuted_input(7 downto 4) <= "0011"; -
- 3
else permuted_input(7 downto 4) <= "1111"; -- F
end if;
end process;
--lookup table t3
q1_t3 : process (befor_t3,permuted_input)
begin
if befor_t3 = "0000" then permuted_input(3 downto 0) <= "1011"; -- B
elsif befor_t3 = "0001" then permuted_input(3 downto 0) <= "1001"; --
9
elsif befor_t3 = "0010" then permuted_input(3 downto 0) <= "0101"; --
5
elsif befor_t3 = "0011" then permuted_input(3 downto 0) <= "0001"; --
1
elsif befor_t3 = "0100" then permuted_input(3 downto 0) <= "1100"; --
C
elsif befor_t3 = "0101" then permuted_input(3 downto 0) <= "0011"; --
3
elsif befor_t3 = "0110" then permuted_input(3 downto 0) <= "1101"; --
D
elsif befor_t3 = "0111" then permuted_input(3 downto 0) <= "1110"; --
E
elsif befor_t3 = "1000" then permuted_input(3 downto 0) <= "0110"; --
6
elsif befor_t3 = "1001" then permuted_input(3 downto 0) <= "0100"; --
4
elsif befor_t3 = "1010" then permuted_input(3 downto 0) <= "0111"; --
7
elsif befor_t3 = "1011" then permuted_input(3 downto 0) <= "1111"; --
F
elsif befor_t3 = "1100" then permuted_input(3 downto 0) <= "0010"; --
2
elsif befor_t3 = "1101" then permuted_input(3 downto 0) <= "0000"; --
0
elsif befor_t3 = "1110" then permuted_input(3 downto 0) <= "1000"; --
8
```

```
else permuted_input(3 downto 0) <= "1010"; -- A
end if;
end process;
--assigning value to the output
-- output = b&a (see page 10 of twofish)
output <= permuted_input(3 downto 0) & permuted_input(7 downto 4); --
---MAKE SURE THIS WORKS WELL!!!!
end struct;
```

21. PHT

```
------------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                         ---
--- ID :  1669                                                 ---
--- Component : This component is the PHT.                     ---
--- Version : 1.0 - Beta                                       ---
------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

-- Pseudo-Hadamard Transform:
--
-- A_in --- + --------------> A_out = A_in + B_in
--          |                 |
--          |                 |
--          |                 |
--          |                 |
--          B_in ------------> + ----> B_out = A_in + 2B_in
--
-- All inputs and outputs are 32-bits wide. Additions
-- are done in modulo 2^32.
--
entity pht is
port       (A_in : in std_logic_vector(31 downto 0);
            B_in : in std_logic_vector(31 downto 0);
            A_out : out std_logic_vector(31 downto 0);
            B_out : out std_logic_vector(31 downto 0));
end pht;

-- Structural implementation
--
-- A_in ----------
--                 | ----> A_out
-- B_in ----------
--
-- A_in ----------
--                 | ----> B_out
-- B_in --- << ---
--
architecture struct OF pht is
component adder32
   port( dataa, datab : in  std_logic_vector(31 downto 0) ;
         cout : out std_logic ;
         cin  : in std_logic:= '0';
         result : out std_logic_vector(31 downto 0)) ;
end component;

signal B_in_x2 : std_logic_vector (31 downto 0);
begin
```

```
-- top adder: A_out = A_in + B_in
top_adder : adder32
port map (dataa => A_in,
          datab => B_in,
          result => A_out);

-- bottom adder: A_out = A_in + 2B_in
bottom_adder : adder32
port map (dataa => A_in,
          datab => B_in_x2,
          result => B_out);

-- Compute 2B_in by shifting B_in left by one.
two_times_2B_in: process (B_in,B_in_x2)
begin
for i IN 31 downto 1 loop
      B_in_x2(i) <= B_in(i-1);
end loop;
B_in_x2(0) <= '0';
end process two_times_2B_in;
end struct;
```

22. REG01

```
------------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                         ---
--- ID :  1669                                                 ---
--- Component : This component is the 1bit register.           ---
--- Version : 1.0 - Beta                                       ---
------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity reg01 is
port (clock, clr, data, enable : in  std_logic;
      q : out std_logic);
end reg01;
architecture dataflow of reg01 is
begin
process (clock,clr)
begin
if (clr ='1') then
    q <='0';
elsif (clock'event and clock='1') then
   if (enable='1') then
          q <= data;
      end if;
end if;
end process;
end dataflow;
```

23. REG16

```
------------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                         ---
--- ID :  1669                                                 ---
--- Component : This component is the 32bit register.          ---
--- Version : 1.0 - Beta                                       ---
------------------------------------------------------------------
```

```
library ieee;
use ieee.std_logic_1164.all;

entity reg16 is
  port(data : in std_logic_vector (15 downto 0);
       clock,clr,enable : in  std_logic;
       q : out std_logic_vector (15 downto 0));
end reg16;

architecture behavior of reg16 is
component reg01
port(clock,clr,enable, data : in  std_logic;
       q : out std_logic);
end component;

begin
aa: for i in 0 to 15 generate
      bb: reg01 port map(clock,clr,enable,data(i),q(i));
    end generate;
end behavior;
```

24. REG32

```
-------------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                          ---
--- ID :  1669                                                  ---
--- Component : This component is the 32bit register.           ---
--- Version : 1.0 - Beta                                        ---
-------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity reg32 is
  port(data : in std_logic_vector (31 downto 0);
       clock,clr,enable : in  std_logic;
       q : out std_logic_vector (31 downto 0));
end reg32;

architecture behavior of reg32 is
component reg01
port(clock,clr,enable, data : in  std_logic;
       q : out std_logic);
end component;

begin
aa: for i in 0 to 31 generate
      bb: reg01 port map(clock,clr,enable,data(i),q(i));
    end generate;
end behavior;
```

25. REG128

```
-------------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                          ---
--- ID :  1669                                                  ---
--- Component : This component is the 128bit register.          ---
--- Version : 1.0 - Beta                                        ---
-------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity reg128 is
  port(clock,clr,enable : in  std_logic;
       data : in std_logic_vector (127 downto 0);
       q : out std_logic_vector (127 downto 0));
end reg128;

architecture behavior of reg128 is
component reg01
port(clock,clr,enable,data : in  std_logic;
       q : out std_logic);
end component;

begin
aa: for i in 0 to 127 generate
      bb: reg01 port map(clock,clr,enable,data(i),q(i));
    end generate;
end behavior;
```

26. ROL1_32

```
-------------------------------------------------------------------------
--
--- Author : Ananda Raja A/L Dore Raja                              --
--- ID :  1669                                                      --
--- Component : This component rotates to the left by 1bit.(32bit) --
--- Version : 1.0 - Beta                                            --
-------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity rol1_32 is
port (data : in std_logic_vector(31 downto 0);
      q : out std_logic_vector(31 downto 0));
end rol1_32;

architecture behavior of rol1_32 is
begin
reg: process(data)
begin
      q <= data(30 downto 0) & data(31); --rotate left 1 bit
end process;
end behavior;
```

27. ROL8_32

```
-------------------------------------------------------------------------
--
--- Author : Ananda Raja A/L Dore Raja                              --
--- ID :  1669                                                      --
---- Component : This component rotates to the left by 8 bits.32bit)-
--- Version : 1.0 - Beta                                            --
-------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
```

```
use ieee.std_logic_arith.all;

entity rol8_32 is
port (data : in std_logic_vector(31 downto 0);
      q : out std_logic_vector(31 downto 0));
end rol8_32;

architecture behavior of rol8_32 is
begin
reg: process(data)
begin
      q <= data(23 downto 0) & data(31 downto 24); --rotate left 8
bit
end process;
end behavior;
```

28. ROL9_32

```
-------------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                          --
--- ID :  1669                                                  --
--- Component : This component rotates to the left by 9 bits.(32bit)-
--- Version : 1.0 - Beta                                        --
-------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity rol9_32 is
port (data : in std_logic_vector(31 downto 0);
      q : out std_logic_vector(31 downto 0));
end rol9_32;

architecture behavior of rol9_32 is
begin
reg: process(data)
begin
      q <= data(22 downto 0) & data(31 downto 23); --rotate left 9
bit
end process;
end behavior;
```

29. ROR1_32

```
-------------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                          --
--- ID :  1669                                                  --
--- Component : This component rotates to the right by 1 bit.(32bit)-
--- Version : 1.0 - Beta                                        ---
-------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity ror1_32 is
port (data : in std_logic_vector(31 downto 0);
      q : out std_logic_vector(31 downto 0));
end ror1_32;

architecture behavior of ror1_32 is
begin
reg: process(data)
begin
      q <= data(0) & data(31 downto 1); --rotate right 1 bit
end process;
end behavior;
```

30. RSMATRIX

```
-------------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                          --
--- ID :  1669                                                  --
--- Component : This component is the RS Matrix.                --
--- Version : 1.0 - Beta                                        --
-------------------------------------------------------------------


--
-- |s0| |01 A4 55 87 5A 58 DB 9E| |m0|
-- |s1| |A4 56 82 F3 1E C6 68 E5||m1|
-- |s2| |02 A1 FC C1 47 AE 3D 19| |m2|
-- |s3| |A4 55 87 5A 58 DB 9E 03| |m3|
-- |m4|
-- |m5|
-- |m6|
-- |m7|
--
-- In Galois field GF(2^8) with primitive polynomial
-- x^8 + x^6 + x^3 + x^2 + 1.
--
--
library ieee;
use ieee.std_logic_1164.all;
entity rsmatrix is
port(m0 : in std_logic_vector(7 downto 0);
     m1 : in std_logic_vector(7 downto 0);
     m2 : in std_logic_vector(7 downto 0);
     m3 : in std_logic_vector(7 downto 0);
     m4 : in std_logic_vector(7 downto 0);
     m5 : in std_logic_vector(7 downto 0);
     m6 : in std_logic_vector(7 downto 0);
     m7 : in std_logic_vector(7 downto 0);
     s0 : out std_logic_vector(7 downto 0);
     s1 : out std_logic_vector(7 downto 0);
     s2 : out std_logic_vector(7 downto 0);
     s3 : out std_logic_vector(7 downto 0));
end rsmatrix;

architecture struct of rsmatrix is

component multgf_A4
port(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end component;

component multgf_55
port(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end component;
```

```
component multgf_87
port(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end component;

component multgf_5A
port(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end component;

component multgf_58
port(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end component;

component multgf_DB
port(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end component;

component multgf_9E
port(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end component;

component multgf_56
port(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end component;

component multgf_82
port(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end component;

component multgf_F3
port(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end component;

component multgf_1E
port(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end component;

component multgf_C6
port(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end component;

component multgf_68
port(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end component;

component multgf_E5
port(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end component;

component multgf_02
port(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end component;

component multgf_A1
port(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end component;

component multgf_FC
port(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end component;

component multgf_C1
port(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end component;

component multgf_47
port(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end component;

component multgf_AE
port(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end component;

component multgf_3D
port(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end component;

component multgf_19
port(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end component;

component multgf_03
port(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end component;

component xor_8x8
port(A : in std_logic_vector(7 downto 0);
B : in std_logic_vector(7 downto 0);
C : in std_logic_vector(7 downto 0);
D : in std_logic_vector(7 downto 0);
E : in std_logic_vector(7 downto 0);
F : in std_logic_vector(7 downto 0);
G : in std_logic_vector(7 downto 0);
H : in std_logic_vector(7 downto 0);
result : out std_logic_vector(7 downto 0));
end component;
SIGNAL m0x01, m1xA4, m2x55, m3x87, m4x5A, m5x58, m6xDB, m7x9E,
m0xA4, m1x56, m2x82, m3xF3, m4x1E, m5xC6, m6x68, m7xE5,
m0x02, m1xA1, m2xFC, m3xC1, m4x47, m5xAE, m6x3D, m7x19,
m1x55, m2x87, m3x5A, m4x58, m5xDB, m6x9E, m7x03
: STD_LOGIC_VECTOR(7 downto 0);
begin
```

```
--
-- compute first row
--
m0x01 <= m0;
m1xA4_comp: multgf_A4 PORT MAP (inbyte => m1, outbyte => m1xA4);
m2x55_comp: multgf_55 PORT MAP (inbyte => m2, outbyte => m2x55);
m3x87_comp: multgf_87 PORT MAP (inbyte => m3, outbyte => m3x87);
m4x5A_comp: multgf_5A PORT MAP (inbyte => m4, outbyte => m4x5A);
m5x58_comp: multgf_58 PORT MAP (inbyte => m5, outbyte => m5x58);
m6xDB_comp: multgf_DB PORT MAP (inbyte => m6, outbyte => m6xDB);
m7x9E_comp: multgf_9E PORT MAP (inbyte => m7, outbyte => m7x9E);
--
-- computer second row
--
m0xA4_comp: multgf_A4 PORT MAP (inbyte => m0, outbyte => m0xA4);
m1x56_comp: multgf_56 PORT MAP (inbyte => m1, outbyte => m1x56);
m2x82_comp: multgf_82 PORT MAP (inbyte => m2, outbyte => m2x82);
m3xF3_comp: multgf_F3 PORT MAP (inbyte => m3, outbyte => m3xF3);
m4x1E_comp: multgf_1E PORT MAP (inbyte => m4, outbyte => m4x1E);
m5xC6_comp: multgf_C6 PORT MAP (inbyte => m5, outbyte => m5xC6);
m6x68_comp: multgf_68 PORT MAP (inbyte => m6, outbyte => m6x68);
m7xE5_comp: multgf_E5 PORT MAP (inbyte => m7, outbyte => m7xE5);
--
-- computer third row
--
m0x02_comp: multgf_02 PORT MAP (inbyte => m0, outbyte => m0x02);
m1xA1_comp: multgf_A1 PORT MAP (inbyte => m1, outbyte => m1xA1);
m2xFC_comp: multgf_FC PORT MAP (inbyte => m2, outbyte => m2xFC);
m3xC1_comp: multgf_C1 PORT MAP (inbyte => m3, outbyte => m3xC1);
m4x47_comp: multgf_47 PORT MAP (inbyte => m4, outbyte => m4x47);
m5xAE_comp: multgf_AE PORT MAP (inbyte => m5, outbyte => m5xAE);
m6x3D_comp: multgf_3D PORT MAP (inbyte => m6, outbyte => m6x3D);
m7x19_comp: multgf_19 PORT MAP (inbyte => m7, outbyte => m7x19);
--
-- computer fourth row
--
-- m0xA4 is already done
m1x55_comp: multgf_55 PORT MAP (inbyte => m1, outbyte => m1x55);
m2x87_comp: multgf_87 PORT MAP (inbyte => m2, outbyte => m2x87);
m3x5A_comp: multgf_5A PORT MAP (inbyte => m3, outbyte => m3x5A);
m4x58_comp: multgf_58 PORT MAP (inbyte => m4, outbyte => m4x58);
m5xDB_comp: multgf_DB PORT MAP (inbyte => m5, outbyte => m5xDB);
m6x9E_comp: multgf_9E PORT MAP (inbyte => m6, outbyte => m6x9E);
m7x03_comp: multgf_03 PORT MAP (inbyte => m7, outbyte => m7x03);
--
-- add elements of row 1
--
row1_xor: xor_8x8
PORT MAP (  A => m0x01,
            B => m1xA4,
            C => m2x55,
            D => m3x87,
            E => m4x5A,
            F => m5x58,
            G => m6xDB,
            H => m7x9E,
            result => s0 );
--
-- add elements of row 2
--
row2_xor: xor_8x8
PORT MAP (  A => m0xA4,
            B => m1x56,
            C => m2x82,
            D => m3xF3,
            E => m4x1E,
            F => m5xC6,
            G => m6x68,
            H => m7xE5,
            result => s1 );
--
-- add elements of row 3
--
row3_xor: xor_8x8
PORT MAP (  A => m0x02,
            B => m1xA1,
            C => m2xFC,
            D => m3xC1,
            E => m4x47,
            F => m5xAE,
            G => m6x3D,
            H => m7x19,
            result => s2 );
--
-- add elements of row 4
--
row4_xor: xor_8x8
PORT MAP (  A => m0xA4,
            B => m1x55,
            C => m2x87,
            D => m3x5A,
            E => m4x58,
            F => m5xDB,
            G => m6x9E,
            H => m7x03,
            result => s3 );
end struct;
--
-- The following multiplications are carriend out
-- in the Galois field GF(2^8) with primitve polynomial
-- x^8 + x^6 + x^3 + x^2 + 1.
--
-- The number is in hexadecimal. inbyte is the input
-- vector and outbyte the output vector.
--
--
-- A4
--
library ieee;
use ieee.std_logic_1164.all;

entity multgf_A4 is
PORT( inbyte : in std_logic_vector(7 downto 0);
      outbyte : out std_logic_vector(7 downto 0));
end multgf_A4;

architecture struct of multgf_A4 is
begin
outbyte(7) <= inbyte(6) XOR inbyte(0);
outbyte(6) <= inbyte(5);
outbyte(5) <= inbyte(6) XOR inbyte(4) XOR inbyte(0);
outbyte(4) <= inbyte(5) XOR inbyte(3);
```

```
outbyte(3) <= inbyte(7) XOR inbyte(4) XOR inbyte(2);
outbyte(2) <= inbyte(7) XOR inbyte(3) XOR inbyte(1) XOR inbyte(0);
outbyte(1) <= inbyte(2);
outbyte(0) <= inbyte(7) XOR inbyte(1);
end struct;
--
-- 55
--
library ieee;
use ieee.std_logic_1164.all;

entity multgf_55 is
PORT( inbyte : in std_logic_vector(7 downto 0);
      outbyte : out std_logic_vector(7 downto 0));
end multgf_55;

architecture struct of multgf_55 is
begin
outbyte(7) <= inbyte(7) XOR inbyte(6) XOR inbyte(5) XOR inbyte(1);
outbyte(6) <= inbyte(7) XOR inbyte(6) XOR inbyte(5) XOR inbyte(4) XOR
inbyte(0);
outbyte(5) <= inbyte(7) XOR inbyte(4) XOR inbyte(3) XOR inbyte(1);
outbyte(4) <= inbyte(7) XOR inbyte(6) XOR inbyte(3) XOR inbyte(2) XOR
inbyte(0);
outbyte(3) <= inbyte(6) XOR inbyte(5) XOR inbyte(2) XOR inbyte(1);
outbyte(2) <= inbyte(7) XOR inbyte(6) XOR inbyte(4) XOR inbyte(0);
outbyte(1) <= inbyte(7) XOR inbyte(3) XOR inbyte(1);
outbyte(0) <= inbyte(7) XOR inbyte(6) XOR inbyte(2) XOR inbyte(0);
end struct;
--
-- 87
--
library ieee;
use ieee.std_logic_1164.all;

entity multgf_87 is
PORT( inbyte : in std_logic_vector(7 downto 0);
      outbyte : out std_logic_vector(7 downto 0));
end multgf_87;

architecture struct of multgf_87 is
begin
outbyte(7) <= inbyte(6) XOR inbyte(4) XOR inbyte(2) XOR inbyte(0);
outbyte(6) <= inbyte(7) XOR inbyte(5) XOR inbyte(3) XOR inbyte(1);
outbyte(5) <= inbyte(7);
outbyte(4) <= inbyte(6);
outbyte(3) <= inbyte(7) XOR inbyte(5);
outbyte(2) <= inbyte(2) XOR inbyte(0);
outbyte(1) <= inbyte(6) XOR inbyte(4) XOR inbyte(2) XOR inbyte(1) XOR
inbyte(0);
outbyte(0) <= inbyte(7) XOR inbyte(5) XOR inbyte(3) XOR inbyte(1) XOR
inbyte(0);
end struct;
--
-- 5A
--
library ieee;
use ieee.std_logic_1164.all;

entity multgf_5A is
PORT(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end multgf_5A;

architecture struct of multgf_5A is
begin
outbyte(7) <= inbyte(7) XOR inbyte(6) XOR inbyte(4) XOR inbyte(1);
outbyte(6) <= inbyte(7) XOR inbyte(6) XOR inbyte(5) XOR inbyte(3) XOR
inbyte(0);
outbyte(5) <= inbyte(5) XOR inbyte(2) XOR inbyte(1);
outbyte(4) <= inbyte(7) XOR inbyte(4) XOR inbyte(1) XOR inbyte(0);
outbyte(3) <= inbyte(7) XOR inbyte(6) XOR inbyte(3) XOR inbyte(0);
outbyte(2) <= inbyte(5) XOR inbyte(4) XOR inbyte(2) XOR inbyte(1);
outbyte(1) <= inbyte(6) XOR inbyte(3) XOR inbyte(0);
outbyte(0) <= inbyte(7) XOR inbyte(5) XOR inbyte(2);
end struct;
--
-- 58
--
library ieee;
use ieee.std_logic_1164.all;

entity multgf_58 is
PORT( inbyte : in std_logic_vector(7 downto 0);
      outbyte : out std_logic_vector(7 downto 0));
end multgf_58;

architecture struct of multgf_58 is
begin
outbyte(7) <= inbyte(7) XOR inbyte(4) XOR inbyte(1);
outbyte(6) <= inbyte(6) XOR inbyte(3) XOR inbyte(0);
outbyte(5) <= inbyte(5) XOR inbyte(4) XOR inbyte(2) XOR inbyte(1);
outbyte(4) <= inbyte(7) XOR inbyte(4) XOR inbyte(3) XOR inbyte(1) XOR
inbyte(0);
outbyte(3) <= inbyte(6) XOR inbyte(3) XOR inbyte(2) XOR inbyte(0);
outbyte(2) <= inbyte(7) XOR inbyte(5) XOR inbyte(4) XOR inbyte(2);
outbyte(1) <= inbyte(6) XOR inbyte(3);
outbyte(0) <= inbyte(5) XOR inbyte(2);
end struct;
--
-- DB
--
library ieee;
use ieee.std_logic_1164.all;

entity multgf_DB is
PORT( inbyte : in std_logic_vector(7 downto 0);
      outbyte : out std_logic_vector(7 downto 0));
end multgf_DB;

architecture struct of multgf_DB is
begin
outbyte(7) <= inbyte(6) XOR inbyte(5) XOR inbyte(2) XOR inbyte(1) XOR
inbyte(0);
outbyte(6) <= inbyte(7) XOR inbyte(5) XOR inbyte(4) XOR inbyte(1) XOR
inbyte(0);
outbyte(5) <= inbyte(7) XOR inbyte(5) XOR inbyte(4) XOR inbyte(3) XOR
inbyte(2) XOR inbyte(1);
outbyte(4) <= inbyte(6) XOR inbyte(4) XOR inbyte(3) XOR inbyte(2) XOR
inbyte(1) XOR inbyte(0);
outbyte(3) <= inbyte(5) XOR inbyte(3) XOR inbyte(2) XOR inbyte(1) XOR
inbyte(0);
```

```
outbyte(2) <= inbyte(7) XOR inbyte(6) XOR inbyte(5) XOR inbyte(4);
outbyte(1) <= inbyte(7) XOR inbyte(4) XOR inbyte(3) XOR inbyte(2) XOR
inbyte(1) XOR inbyte(0);
outbyte(0) <= inbyte(7) XOR inbyte(6) XOR inbyte(3) XOR inbyte(2) XOR
inbyte(1) XOR inbyte(0);
end struct;
--
-- 9E
--
library ieee;
use ieee.std_logic_1164.all;

entity multgf_9E is
PORT( inbyte : in std_logic_vector(7 downto 0);
      outbyte : out std_logic_vector(7 downto 0));
end multgf_9E;

architecture struct of multgf_9E is
begin
outbyte(7) <= inbyte(5) XOR inbyte(3) XOR inbyte(2) XOR inbyte(0);
outbyte(6) <= inbyte(7) XOR inbyte(4) XOR inbyte(2) XOR inbyte(1);
outbyte(5) <= inbyte(7) XOR inbyte(6) XOR inbyte(5) XOR inbyte(2) XOR
inbyte(1);
outbyte(4) <= inbyte(6) XOR inbyte(5) XOR inbyte(4) XOR inbyte(1) XOR
inbyte(0);
outbyte(3) <= inbyte(5) XOR inbyte(4) XOR inbyte(3) XOR inbyte(0);
outbyte(2) <= inbyte(5) XOR inbyte(4) XOR inbyte(0);
outbyte(1) <= inbyte(7) XOR inbyte(5) XOR inbyte(4) XOR inbyte(2) XOR
inbyte(0);
outbyte(0) <= inbyte(6) XOR inbyte(4) XOR inbyte(3) XOR inbyte(1);
end struct;
--
-- 56
--
library ieee;
use ieee.std_logic_1164.all;

entity multgf_56 is
PORT( inbyte : in std_logic_vector(7 downto 0);
      outbyte : out std_logic_vector(7 downto 0));
end multgf_56;

architecture struct of multgf_56 is
begin
outbyte(7) <= inbyte(5) XOR inbyte(1);
outbyte(6) <= inbyte(4) XOR inbyte(0);
outbyte(5) <= inbyte(7) XOR inbyte(5) XOR inbyte(3) XOR inbyte(1);
outbyte(4) <= inbyte(7) XOR inbyte(6) XOR inbyte(4) XOR inbyte(2) XOR
inbyte(0);
outbyte(3) <= inbyte(7) XOR inbyte(6) XOR inbyte(5) XOR inbyte(3) XOR
inbyte(1);
outbyte(2) <= inbyte(6) XOR inbyte(4) XOR inbyte(2) XOR inbyte(1) XOR
inbyte(0);
outbyte(1) <= inbyte(7) XOR inbyte(3) XOR inbyte(0);
outbyte(0) <= inbyte(6) XOR inbyte(2);
end struct;
--
-- 82
--
library ieee;
use ieee.std_logic_1164.all;

entity multgf_82 is
PORT( inbyte : in std_logic_vector(7 downto 0);
      outbyte : out std_logic_vector(7 downto 0));
end multgf_82;

architecture struct of multgf_82 is
begin
outbyte(7) <= inbyte(6) XOR inbyte(5) XOR inbyte(4) XOR inbyte(2) XOR
inbyte(0);
outbyte(6) <= inbyte(7) XOR inbyte(5) XOR inbyte(4) XOR inbyte(3) XOR
inbyte(1);
outbyte(5) <= inbyte(7) XOR inbyte(5) XOR inbyte(3);
outbyte(4) <= inbyte(7) XOR inbyte(6) XOR inbyte(4) XOR inbyte(2);
outbyte(3) <= inbyte(6) XOR inbyte(5) XOR inbyte(3) XOR inbyte(1);
outbyte(2) <= inbyte(6);
outbyte(1) <= inbyte(7) XOR inbyte(6) XOR inbyte(4) XOR inbyte(2) XOR
inbyte(0);
outbyte(0) <= inbyte(7) XOR inbyte(6) XOR inbyte(5) XOR inbyte(3) XOR
inbyte(1);
end struct;
--
-- F3
--
library ieee;
use ieee.std_logic_1164.all;

entity multgf_F3 is
PORT( inbyte : in std_logic_vector(7 downto 0);
      outbyte : out std_logic_vector(7 downto 0));
end multgf_F3;

architecture struct of multgf_F3 is
begin
outbyte(7) <= inbyte(7) XOR inbyte(6) XOR inbyte(5) XOR inbyte(1) XOR
inbyte(0);
outbyte(6) <= inbyte(6) XOR inbyte(5) XOR inbyte(4) XOR inbyte(0);
outbyte(5) <= inbyte(7) XOR inbyte(6) XOR inbyte(4) XOR inbyte(3) XOR
inbyte(1) XOR inbyte(0);
outbyte(4) <= inbyte(7) XOR inbyte(6) XOR inbyte(5) XOR inbyte(3) XOR
inbyte(2) XOR inbyte(0);
outbyte(3) <= inbyte(6) XOR inbyte(5) XOR inbyte(4) XOR inbyte(2) XOR
inbyte(1);
outbyte(2) <= inbyte(7) XOR inbyte(6) XOR inbyte(4) XOR inbyte(3);
outbyte(1) <= inbyte(7) XOR inbyte(3) XOR inbyte(2) XOR inbyte(1) XOR
inbyte(0);
outbyte(0) <= inbyte(7) XOR inbyte(6) XOR inbyte(2) XOR inbyte(1) XOR
inbyte(0);
end struct;
--
-- 1E
--
library ieee;
use ieee.std_logic_1164.all;

entity multgf_1E is
PORT( inbyte : in std_logic_vector(7 downto 0);
      outbyte : out std_logic_vector(7 downto 0));
end multgf_1E;

architecture struct of multgf_1E is
```

```
begin
outbyte(7) <= inbyte(4) XOR inbyte(3);
outbyte(6) <= inbyte(7) XOR inbyte(3) XOR inbyte(2);
outbyte(5) <= inbyte(6) XOR inbyte(4) XOR inbyte(3) XOR inbyte(2) XOR
inbyte(1);
outbyte(4) <= inbyte(7) XOR inbyte(5) XOR inbyte(3) XOR inbyte(2) XOR
inbyte(1) XOR inbyte(0);
outbyte(3) <= inbyte(7) XOR inbyte(6) XOR inbyte(4) XOR inbyte(2) XOR
inbyte(1) XOR inbyte(0);
outbyte(2) <= inbyte(7) XOR inbyte(6) XOR inbyte(5) XOR inbyte(4) XOR
inbyte(1) XOR inbyte(0);
outbyte(1) <= inbyte(6) XOR inbyte(5) XOR inbyte(0);
outbyte(0) <= inbyte(5) XOR inbyte(4);
end struct;
--
-- C6
--
library ieee;
use ieee.std_logic_1164.all;

entity multgf_C6 is
PORT( inbyte : in std_logic_vector(7 downto 0);
      outbyte : out std_logic_vector(7 downto 0));
end multgf_C6;

architecture struct of multgf_C6 is
begin
outbyte(7) <= inbyte(7) XOR inbyte(5) XOR inbyte(4) XOR inbyte(3) XOR
inbyte(2) XOR inbyte(1) XOR inbyte(0);
outbyte(6) <= inbyte(7) XOR inbyte(6) XOR inbyte(4) XOR inbyte(3) XOR
inbyte(2) XOR inbyte(1) XOR inbyte(0);
outbyte(5) <= inbyte(7) XOR inbyte(6) XOR inbyte(4);
outbyte(4) <= inbyte(7) XOR inbyte(6) XOR inbyte(5) XOR inbyte(3);
outbyte(3) <= inbyte(6) XOR inbyte(5) XOR inbyte(4) XOR inbyte(2);
outbyte(2) <= inbyte(7) XOR inbyte(2) XOR inbyte(0);
outbyte(1) <= inbyte(7) XOR inbyte(6) XOR inbyte(5) XOR inbyte(4) XOR
inbyte(3) XOR inbyte(2) XOR inbyte(0);
outbyte(0) <= inbyte(6) XOR inbyte(5) XOR inbyte(4) XOR inbyte(3) XOR
inbyte(2) XOR inbyte(1);
end struct;
--
-- 68
--
library ieee;
use ieee.std_logic_1164.all;

entity multgf_68 is
PORT( inbyte : in std_logic_vector(7 downto 0);
      outbyte : out std_logic_vector(7 downto 0));
end multgf_68;

architecture struct of multgf_68 is
begin
outbyte(7) <= inbyte(7) XOR inbyte(6) XOR inbyte(5) XOR inbyte(3) XOR
inbyte(2) XOR inbyte(1);
outbyte(6) <= inbyte(7) XOR inbyte(6) XOR inbyte(5) XOR inbyte(4) XOR
inbyte(2) XOR inbyte(1) XOR inbyte(0);
outbyte(5) <= inbyte(4) XOR inbyte(2) XOR inbyte(0);
outbyte(4) <= inbyte(3) XOR inbyte(1);
outbyte(3) <= inbyte(7) XOR inbyte(2) XOR inbyte(0);
outbyte(2) <= inbyte(7) XOR inbyte(5) XOR inbyte(3) XOR inbyte(2);
outbyte(1) <= inbyte(7) XOR inbyte(5) XOR inbyte(4) XOR inbyte(3);
outbyte(0) <= inbyte(7) XOR inbyte(6) XOR inbyte(4) XOR inbyte(3) XOR
inbyte(2);
end struct;
--
-- E5
--
library ieee;
use ieee.std_logic_1164.all;

entity multgf_E5 is
PORT( inbyte : in std_logic_vector(7 downto 0);
      outbyte : out std_logic_vector(7 downto 0));
end multgf_E5;

architecture struct of multgf_E5 is
begin
outbyte(7) <= inbyte(7) XOR inbyte(5) XOR inbyte(3) XOR inbyte(1) XOR
inbyte(0);
outbyte(6) <= inbyte(6) XOR inbyte(4) XOR inbyte(2) XOR inbyte(0);
outbyte(5) <= inbyte(0);
outbyte(4) <= inbyte(7);
outbyte(3) <= inbyte(6);
outbyte(2) <= inbyte(3) XOR inbyte(1) XOR inbyte(0);
outbyte(1) <= inbyte(7) XOR inbyte(5) XOR inbyte(3) XOR inbyte(2) XOR
inbyte(1);
outbyte(0) <= inbyte(6) XOR inbyte(4) XOR inbyte(2) XOR inbyte(1) XOR
inbyte(0);
end struct;
--
-- 02
--
library ieee;
use ieee.std_logic_1164.all;

entity multgf_02 is
PORT( inbyte : in std_logic_vector(7 downto 0);
      outbyte : out std_logic_vector(7 downto 0));
end multgf_02;

architecture struct of multgf_02 is
begin
outbyte(7) <= inbyte(6);
outbyte(6) <= inbyte(7) XOR inbyte(5);
outbyte(5) <= inbyte(4);
outbyte(4) <= inbyte(3);
outbyte(3) <= inbyte(7) XOR inbyte(2);
outbyte(2) <= inbyte(7) XOR inbyte(1);
outbyte(1) <= inbyte(0);
outbyte(0) <= inbyte(7);
end struct;
--
-- A1
--
library ieee;
use ieee.std_logic_1164.all;

entity multgf_A1 is
PORT( inbyte : in std_logic_vector(7 downto 0);
      outbyte : out std_logic_vector(7 downto 0));
end multgf_A1;
```

```
architecture struct of multgf_A1 is
begin
outbyte(7) <= inbyte(6) XOR inbyte(5) XOR inbyte(0);
outbyte(6) <= inbyte(5) XOR inbyte(4);
outbyte(5) <= inbyte(6) XOR inbyte(5) XOR inbyte(4) XOR inbyte(3) XOR
inbyte(0);
outbyte(4) <= inbyte(7) XOR inbyte(5) XOR inbyte(4) XOR inbyte(3) XOR
inbyte(2);
outbyte(3) <= inbyte(6) XOR inbyte(4) XOR inbyte(3) XOR inbyte(2) XOR
inbyte(1);
outbyte(2) <= inbyte(7) XOR inbyte(6) XOR inbyte(3) XOR inbyte(2) XOR
inbyte(1);
outbyte(1) <= inbyte(7) XOR inbyte(2) XOR inbyte(1);
outbyte(0) <= inbyte(7) XOR inbyte(6) XOR inbyte(1) XOR inbyte(0);
end struct;
--
-- FC
--
library ieee;
use ieee.std_logic_1164.all;

entity multgf_FC is
PORT( inbyte : in std_logic_vector(7 downto 0);
      outbyte : out std_logic_vector(7 downto 0));
end multgf_FC;

architecture struct of multgf_FC is
begin
outbyte(7) <= inbyte(7) XOR inbyte(6) XOR inbyte(4) XOR inbyte(1) XOR
inbyte(0);
outbyte(6) <= inbyte(6) XOR inbyte(5) XOR inbyte(3) XOR inbyte(0);
outbyte(5) <= inbyte(6) XOR inbyte(5) XOR inbyte(2) XOR inbyte(1) XOR
inbyte(0);
outbyte(4) <= inbyte(7) XOR inbyte(5) XOR inbyte(4) XOR inbyte(1) XOR
inbyte(0);
outbyte(3) <= inbyte(7) XOR inbyte(6) XOR inbyte(4) XOR inbyte(3) XOR
inbyte(0);
outbyte(2) <= inbyte(5) XOR inbyte(4) XOR inbyte(3) XOR inbyte(2) XOR
inbyte(1) XOR inbyte(0);
outbyte(1) <= inbyte(6) XOR inbyte(3) XOR inbyte(2);
outbyte(0) <= inbyte(7) XOR inbyte(5) XOR inbyte(2) XOR inbyte(1);
end struct;
--
-- C1
--
library ieee;
use ieee.std_logic_1164.all;

entity multgf_C1 is
PORT( inbyte : in std_logic_vector(7 downto 0);
      outbyte : out std_logic_vector(7 downto 0));
end multgf_C1;

architecture struct of multgf_C1 is
begin
outbyte(7) <= inbyte(7) XOR inbyte(6) XOR inbyte(4) XOR inbyte(3) XOR
inbyte(2) XOR inbyte(1) XOR inbyte(0);
outbyte(6) <= inbyte(6) XOR inbyte(5) XOR inbyte(3) XOR inbyte(2) XOR
inbyte(1) XOR inbyte(0);
outbyte(5) <= inbyte(7) XOR inbyte(6) XOR inbyte(5) XOR inbyte(3);
outbyte(4) <= inbyte(6) XOR inbyte(5) XOR inbyte(4) XOR inbyte(2);
outbyte(3) <= inbyte(5) XOR inbyte(4) XOR inbyte(3) XOR inbyte(1);
outbyte(2) <= inbyte(6) XOR inbyte(1);
outbyte(1) <= inbyte(6) XOR inbyte(5) XOR inbyte(4) XOR inbyte(3) XOR
inbyte(2) XOR inbyte(1);
outbyte(0) <= inbyte(7) XOR inbyte(5) XOR inbyte(4) XOR inbyte(3) XOR
inbyte(2) XOR inbyte(1) XOR inbyte(0);
end struct;
--
-- 47
--
library ieee;
use ieee.std_logic_1164.all;

entity multgf_47 is
PORT( inbyte : in std_logic_vector(7 downto 0);
      outbyte : out std_logic_vector(7 downto 0));
end multgf_47;

architecture struct of multgf_47 is
begin
outbyte(7) <= inbyte(3) XOR inbyte(1);
outbyte(6) <= inbyte(7) XOR inbyte(2) XOR inbyte(0);
outbyte(5) <= inbyte(6) XOR inbyte(3);
outbyte(4) <= inbyte(5) XOR inbyte(2);
outbyte(3) <= inbyte(7) XOR inbyte(4) XOR inbyte(1);
outbyte(2) <= inbyte(6) XOR inbyte(1) XOR inbyte(0);
outbyte(1) <= inbyte(5) XOR inbyte(3) XOR inbyte(1) XOR inbyte(0);
outbyte(0) <= inbyte(4) XOR inbyte(2) XOR inbyte(0);
end struct;
--
-- AE
--
library ieee;
use ieee.std_logic_1164.all;

entity multgf_AE is
PORT( inbyte : in std_logic_vector(7 downto 0);
      outbyte : out std_logic_vector(7 downto 0));
end multgf_AE;

architecture struct of multgf_AE is
begin
outbyte(7) <= inbyte(6) XOR inbyte(4) XOR inbyte(0);
outbyte(6) <= inbyte(5) XOR inbyte(3);
outbyte(5) <= inbyte(7) XOR inbyte(6) XOR inbyte(2) XOR inbyte(0);
outbyte(4) <= inbyte(7) XOR inbyte(6) XOR inbyte(5) XOR inbyte(1);
outbyte(3) <= inbyte(7) XOR inbyte(6) XOR inbyte(5) XOR inbyte(4) XOR
inbyte(0);
outbyte(2) <= inbyte(5) XOR inbyte(3) XOR inbyte(0);
outbyte(1) <= inbyte(6) XOR inbyte(2) XOR inbyte(0);
outbyte(0) <= inbyte(7) XOR inbyte(5) XOR inbyte(1);
end struct;
--
-- 3D
--
library ieee;
use ieee.std_logic_1164.all;

entity multgf_3D is
PORT( inbyte : in std_logic_vector(7 downto 0);
```

```
      outbyte : out std_logic_vector(7 downto 0));
end multgf_3D;

architecture struct of multgf_3D is
begin
outbyte(7) <= inbyte(3) XOR inbyte(2);
outbyte(6) <= inbyte(2) XOR inbyte(1);
outbyte(5) <= inbyte(7) XOR inbyte(3) XOR inbyte(2) XOR inbyte(1) XOR
inbyte(0);
outbyte(4) <= inbyte(7) XOR inbyte(6) XOR inbyte(2) XOR inbyte(1) XOR
inbyte(0);
outbyte(3) <= inbyte(7) XOR inbyte(6) XOR inbyte(5) XOR inbyte(1) XOR
inbyte(0);
outbyte(2) <= inbyte(6) XOR inbyte(5) XOR inbyte(4) XOR inbyte(3) XOR
inbyte(2) XOR inbyte(0);
outbyte(1) <= inbyte(5) XOR inbyte(4) XOR inbyte(1);
outbyte(0) <= inbyte(4) XOR inbyte(3) XOR inbyte(0);
end struct;
--
-- 19
--
library ieee;
use ieee.std_logic_1164.all;

entity multgf_19 is
PORT(inbyte : in std_logic_vector(7 downto 0);
     outbyte : out std_logic_vector(7 downto 0));
end multgf_19;

architecture struct of multgf_19 is
begin
outbyte(7) <= inbyte(6) XOR inbyte(5) XOR inbyte(4) XOR inbyte(3);
outbyte(6) <= inbyte(5) XOR inbyte(4) XOR inbyte(3) XOR inbyte(2);
outbyte(5) <= inbyte(6) XOR inbyte(5) XOR inbyte(2) XOR inbyte(1);
outbyte(4) <= inbyte(5) XOR inbyte(4) XOR inbyte(1) XOR inbyte(0);
outbyte(3) <= inbyte(7) XOR inbyte(4) XOR inbyte(3) XOR inbyte(0);
outbyte(2) <= inbyte(5) XOR inbyte(4) XOR inbyte(2);
outbyte(1) <= inbyte(7) XOR inbyte(6) XOR inbyte(5) XOR inbyte(1);
outbyte(0) <= inbyte(7) XOR inbyte(6) XOR inbyte(5) XOR inbyte(4) XOR
inbyte(0);
end struct;
--
-- 03
--
library ieee;
use ieee.std_logic_1164.all;

entity multgf_03 is
PORT( inbyte : in std_logic_vector(7 downto 0);
      outbyte : out std_logic_vector(7 downto 0));
end multgf_03;
architecture struct of multgf_03 is
begin
outbyte(7) <= inbyte(7) XOR inbyte(6);
outbyte(6) <= inbyte(7) XOR inbyte(6) XOR inbyte(5);
outbyte(5) <= inbyte(5) XOR inbyte(4);
outbyte(4) <= inbyte(4) XOR inbyte(3);
outbyte(3) <= inbyte(7) XOR inbyte(3) XOR inbyte(2);
outbyte(2) <= inbyte(7) XOR inbyte(2) XOR inbyte(1);
outbyte(1) <= inbyte(1) XOR inbyte(0);
outbyte(0) <= inbyte(7) XOR inbyte(0);
end struct;
```

31. S_BOXES

```
----------------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                               --
--- ID :  1669                                                       --
--- Component : This component is the S-boxes.                       --
--- Version : 1.0 - Beta                                             --
----------------------------------------------------------------------

 library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

--                          S0                S1
--                          |                 |
--                          |                 |
-- incoming --> B 1 --> XOR --> BANK2 --> XOR --> BANK3 --> outcome
--
-- Bank1: q0,q1,q0,q1
-- Bank2: q0,q0,q1,q1
-- Bank3: q1,q0,q1,q0
--
--S-boxes used for multiple permutation
entity S_boxes is
port( income : in std_logic_vector(31 downto 0);
            S0 : in std_logic_vector(31 downto 0);
            S1 : in std_logic_vector(31 downto 0);
            outcome : out std_logic_vector(31 downto 0);
            out_bank1_x,
            in_bank2_x,
            out_bank2_x,
            in_bank3_x : out std_logic_vector(31 downto 0));
end S_boxes;

--Building block of the S-boxes
architecture struct OF S_boxes IS
--
-- component declaration
--
--q1 permutation modules
component Perm_q1
port( input :in std_logic_vector(7 downto 0);
            output :out std_logic_vector(7 downto 0));
end component;

--q0 permutation modules
component Perm_q0
port( input :in std_logic_vector(7 downto 0);
            output :out std_logic_vector(7 downto 0));
end component;

--XOR modules
component xor_2x32
PORT(a : in std_logic_vector(31 downto 0);
            b : in std_logic_vector(31 downto 0);
            result : out std_logic_vector(31 downto 0));
end component;
```

```
--SIGNAL DECLARATION
Signal
out_bank1, --Output of the first bank of q-permutations
in_bank2, --Input to the second bank of q-permutations (after XORing
with S0)
out_bank2, --Output of the second bank of q-permutations
in_bank3, --Input to the third bank of q-permutations (after XORing
with S1)
out_bank3 : std_logic_vector(31 downto 0); --Output of the third bank
of q-permutations

--start of s_boxes structure
BEGIN
out_bank1_x <= out_bank1;
in_bank2_x <= in_bank2;
out_bank2_x <= out_bank2;
in_bank3_x <= in_bank3;
--
-- BANK 1
--
--first permutation in S-box 0
S_box0_bank1 : Perm_q0
port map( input => income(7 downto 0),
          output => out_bank1(7 downto 0));
--first permutation in S-box 1
S_box1_bank1 : Perm_q1
port map( input => income(15 downto 8),
          output => out_bank1(15 downto 8));
--first permutation in S-box 2
S_box2_bank1 : Perm_q0
port map( input => income(23 downto 16),
          output => out_bank1(23 downto 16));
--first permutation in S-box 3
S_box3_bank1 : Perm_q1
port map( input => income(31 downto 24),
          output => out_bank1(31 downto 24));

--XOR the converted output of the first bank of permutations and S0
XOR_S0 : xor_2x32
port map( a => out_bank1,
          b => S0,
          result => in_bank2);
--
-- BANK 2
--
--Second permutation in S-box 0
S_box0_bank2 : Perm_q0
port map( input => in_bank2(7 downto 0),
          output => out_bank2(7 downto 0));
--Second permutation in S-box 1
S_box1_bank2 : Perm_q0
port map( input => in_bank2(15 downto 8),
          output => out_bank2(15 downto 8));
--Second permutation in S-box 2
S_box2_bank2 : Perm_q1
port map( input => in_bank2(23 downto 16),
          output => out_bank2(23 downto 16));
--Second permutation in S-box 3
S_box3_bank2 : Perm_q1
port map( input => in_bank2(31 downto 24),
          output => out_bank2(31 downto 24));
--
-- BANK2 XOR S1
--

XOR_S1 : xor_2x32
port map( a => out_bank2,
          b => S1,
          result => in_bank3);
--
-- BANK 3
--
--Third permutation in S-box 0
S_box0_bank3 : Perm_q1
port map( input => in_bank3(7 downto 0),
          output => out_bank3(7 downto 0));
--Third permutation in S-box 1
S_box1_bank3 : Perm_q0
port map( input => in_bank3(15 downto 8),
          output => out_bank3(15 downto 8));
--Third permutation in S-box 2
S_box2_bank3 : Perm_q1
port map( input => in_bank3(23 downto 16),
          output => out_bank3(23 downto 16));
--Thirdd permutation in S-box 3
S_box3_bank3 : Perm_q0
port map( input => in_bank3(31 downto 24),
          output => out_bank3(31 downto 24));
--
-- Final result
--
--Assigning output from signal
outcome <= out_bank3;
end struct;
```

32. SUB_OPSELECT

```
--------------------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                                   --
--- ID :  1669                                                           --
--- Component : This component is the suboperation selector.             --
--- Version : 1.0 - Beta                                                 --
--------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;

entity sub_opselect is
port (rotate_before : in std_logic;
      F_fct : in std_logic_vector(31 downto 0);
      input : in std_logic_vector(31 downto 0);
      output : out std_logic_vector(31 downto 0));
end sub_opselect;

architecture struct of sub_opselect is
component mux_2x32
port (sel : in std_logic;
      in_0 : in std_logic_vector (31 downto 0);
      in_1 : in std_logic_vector (31 downto 0);
      output : out std_logic_vector (31 downto 0));
```

```
end component;

component xor_2x32
port (a : in std_logic_vector(31 downto 0);
      b : in std_logic_vector(31 downto 0);
      result : out std_logic_vector(31 downto 0));
end component;

component rol1_32
port (data : in std_logic_vector(31 downto 0);
      q : out std_logic_vector(31 downto 0));
end component;

component ror1_32
port (data : in std_logic_vector(31 downto 0);
      q : out std_logic_vector(31 downto 0));
end component;

signal rolinput,outmux1,outmux1_xor_f,ror_outmux1_xor_f
                  :    std_logic_vector (31 downto 0 );

begin

rotleft1: rol1_32
port map (data => input,
          q => rolinput);

sel1: mux_2x32
port map(sel => rotate_before,
         in_0 => input,
         in_1 => rolinput,
         output =>outmux1);

xor1: xor_2x32
port map (a => F_fct,
          b => outmux1,
          result => outmux1_xor_f);

rotright1: ror1_32
port map (data => outmux1_xor_f,
          q => ror_outmux1_xor_f);

sel2: mux_2x32
port map( sel => rotate_before,
          in_0 => ror_outmux1_xor_f,
          in_1 => outmux1_xor_f,
          output => output);
end struct;
```

33. WRAPPER

```
----------------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                               --
--- ID :  1669                                                       --
--- Component : This component is the top-level wrapper.             --
--- Version : 1.0 - Beta                                             --
----------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity wrapper is
    Port (  address : in std_logic_vector(4 downto 0); -- address
            write   : in std_logic ; -- write
            read    : in std_logic ; -- read
            clock   : in std_logic ; -- clock
            reset   : in std_logic ; -- reset
            enable  : in std_logic ; -- enable
            load_key: in std_logic ; -- load key
            start   : in std_logic ; -- start process
            encrypt : in std_logic ; -- encrypt or decrypt
            append  : in std_logic ; -- encrypt or decrypt
            indatabus  : in std_logic_vector(31 downto 0) ; -- input
databus
            idle    : out std_logic ; -- idle
       outdatabus  : out std_logic_vector(31 downto 0)) ; --databus

end wrapper;

architecture Behavioral of wrapper is


component core
port (inport : in std_logic_vector(127 downto 0); --input from
cleartext entity
      inkey : in std_logic_vector (127 downto 0); -- input from
keymodule entity
      clk : in std_logic; --Clock signal
      reset : in std_logic;
      usr_ld_key : in std_logic; --Usr requests load key
      usr_start : in std_logic; --Usr requests start
      usr_encrypt : in std_logic; --Usr requests encrypt
      idle : out std_logic; --Device is idle
      outCiphertext : out std_logic_vector(127 downto 0));
  end component;


component reg32
port(clock,clr,enable : in  std_logic;
     data : in std_logic_vector (31 downto 0);
     q : out std_logic_vector (31 downto 0));
end component;


signal
lowerplaintext0,lowerplaintext1,lowerplaintext2,lowerplaintext3,
lowplain0,lowplain1,lowplain2,lowplain3,
lowerkey0,  lowerkey1,  lowerkey2,  lowerkey3,
lowkey0,    lowkey1,    lowkey2,    lowkey3,
lowerciphertext0,lowerciphertext1,lowerciphertext2,lowerciphertext3
: std_logic_vector(31 downto 0);

Signal writesignal,
       readsignal : std_logic;

signal inputplaintext,
       outCiphertext,
       inputkey : std_logic_vector (127 downto 0);
```

```
begin

process (clock)

begin
if (address = "00000") then lowerplaintext0 <= indatabus;
elsif (address = "00001") then lowerplaintext1 <= indatabus;
elsif (address = "00010") then lowerplaintext2 <= indatabus;
elsif (address = "00011") then lowerplaintext3 <= indatabus;

elsif (address = "00100") then lowerkey0 <= indatabus;
elsif (address = "00101") then lowerkey1 <= indatabus;
elsif (address = "00110") then lowerkey2 <= indatabus;
elsif (address = "00111") then lowerkey3 <= indatabus;

elsif (address = "01000") then lowerciphertext0 <= outCiphertext(31
downto 0);
elsif (address = "01001") then lowerciphertext1 <= outCiphertext(63
downto 32);
elsif (address = "01010") then lowerciphertext2 <= outCiphertext(95
downto 64);
elsif (address = "01011") then lowerciphertext3 <= outCiphertext(127
downto 96);

else null;
end if;
end process;

--Signals to enable a process

--Write

process(write,read,enable)
begin
if (write = '1' and read = '0' and enable = '1') then writesignal <=
'1';
else writesignal <= '0';
end if;
end process;

--Read
process(write,read,enable)
begin
if (write = '0' and read = '1' and enable = '1') then readsignal <=
'1';
else readsignal <= '0';
end if;
end process;

--Writing Process

memblock0: reg32
port map (data => lowerplaintext0,
          clock => clock,
          clr => reset,
          enable => writesignal,
          q => lowplain0);

memblock1: reg32
port map (data => lowerplaintext1,
          clock => clock,
          clr => reset,
          enable => writesignal,
          q => lowplain1);

memblock2: reg32
port map (data => lowerplaintext2,
          clock => clock,
          clr => reset,
          enable => writesignal,
          q => lowplain2);

memblock3: reg32
port map (data => lowerplaintext3,
          clock => clock,
          clr => reset,
          enable => writesignal,
          q => lowplain3);

memblock4: reg32
port map (data => lowerkey0,
          clock => clock,
          clr => reset,
          enable => writesignal,
          q => lowkey0);

memblock5: reg32
port map (data => lowerkey1,
          clock => clock,
          clr => reset,
          enable => writesignal,
          q => lowkey1);

memblock6: reg32
port map (data => lowerkey2,
          clock => clock,
          clr => reset,
          enable => writesignal,
          q => lowkey2);

memblock7: reg32
port map (data => lowerkey3,
          clock => clock,
          clr => reset,
          enable => writesignal,
          q => lowkey3);



process(clock)
begin
      if (append = '1')  then
            inputplaintext <= lowplain3 & lowplain2 & lowplain1 &
lowplain0 ;
            inputkey <=  lowkey3 & lowkey2 & lowkey1 & lowkey0;
      else
            inputplaintext <=inputplaintext;
            inputkey <=inputkey;
end if;
end process;
--Reading Process
```

```
process (clock)

begin
if (address = "01000" and readsignal ='1') then outdatabus
<=lowerciphertext0;
elsif (address = "01001" and readsignal ='1') then outdatabus
<=lowerciphertext1;
elsif (address = "01010" and readsignal ='1') then outdatabus
<=lowerciphertext2;
elsif (address = "01011" and readsignal ='1') then outdatabus
<=lowerciphertext3;

else null;
end if;
end process;


maincore : core
port map (inport => inputplaintext,
          inkey => inputkey,
          clk => clock,
          reset => reset,
          usr_ld_key => load_key,
          usr_start => start,
          usr_encrypt => encrypt,
          idle => idle,
          outCiphertext =>outCiphertext);
end Behavioral;
```

34. XOR_2X1

```
----------------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                               --
--- ID :  1669                                                       --
--- Component : This component XORs 2 single bit values.            --
--- Version : 1.0 - Beta                                             --
----------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity xor_2x1 is
port(a,b : in std_logic;
     result : out std_logic);
end xor_2x1;

architecture dataflow of xor_2x1 is
begin
      process(a,b)
      begin
            result <= (a xor b);
      end process;
end dataflow;
```

35. XOR_2X4

```
----------------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                               --
--- ID :  1669                                                       --
--- Component : This component XORs 2 -> 4 bit values.               --
--- Version : 1.0 - Beta                                             --
----------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity xor_2x4 is
PORT(a : in std_logic_vector(3 downto 0);
     b : in std_logic_vector(3 downto 0);
     result : out std_logic_vector(3 downto 0));
end xor_2x4;

architecture behavior of xor_2x4 is
component xor_2x1
PORT(a,b : in std_logic;
     result : out std_logic);
END component;

begin
aa: for i in 0 to 3 generate
      bb: xor_2x1 port map(a(i),b(i),result(i));
    end generate;
end behavior;
```

36. XOR_2X32

```
----------------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                               ---
--- ID :  1669                                                       ---
--- Component : This component XORs 2 -> 32 bit values.              ---
--- Version : 1.0 - Beta                                             ---
----------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity xor_2x32 is
port(a,b : in std_logic_vector(31 downto 0);
     result : out std_logic_vector(31 downto 0));
end xor_2x32;

architecture behavior of xor_2x32 is
component xor_2x1
      PORT(a,b : in std_logic;
           result : out std_logic);
END component;

begin
aa: for i in 0 to 31 generate
      bb: xor_2x1 port map(a(i),b(i),result(i));
    end generate;
end behavior;
```

37. XOR_3X1

```
--------------------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                                   --
---- ID :  1669                                                          --
---- Component : This component XORs 3 single bit values.               --
--- Version : 1.0 - Beta                                                 --
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity xor_3x1 is
port(dataa,datab,datac : in std_logic;
     result : out std_logic);
end xor_3x1;

architecture dataflow of xor_3x1 is
begin
      process(dataa,datab,datac)
      begin
            result <= (dataa xor datab xor datac);
      end process;
end dataflow;
```

38. XOR_3X4

```
--------------------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                                   --
--- ID :  1669                                                          --
--- Component : This component XORs 3 -> 4 bit values.                  --
--- Version : 1.0 - Beta                                                --
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity xor_3x4 is
PORT( dataa : in std_logic_vector(3 downto 0);
      datab : in std_logic_vector(3 downto 0);
      datac : in std_logic_vector(3 downto 0);
      result : out std_logic_vector(3 downto 0));
end xor_3x4;

architecture behavior of xor_3x4 is
component xor_3x1
PORT(dataa,datab,datac : in std_logic;
     result : out std_logic);
END component;

begin
aa: for i in 0 to 3 generate
      bb: xor_3x1 port map(dataa(i),datab(i),datac(i),result(i));
    end generate;
end behavior;
```

39. XOR_4X1

```
--------------------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                                   ---
--- ID :  1669                                                          ---
--- Component : This component XORs 4 single bit values.               ---
--- Version : 1.0 - Beta                                                ---
--------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;

entity xor_4x1 is
port (x,y,z,w : in std_logic;
     result : out std_logic);
end xor_4x1;

architecture dataflow of xor_4x1 is
begin
      process(x,y,z,w)
      begin
            result <= (x xor y xor z xor w);
      end process;
end dataflow;
```

40. XOR_4X8

```
--------------------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                                   ---
--- ID :  1669                                                          ---
--- Component : This component XORs 4 ->  8 bit values.                 ---
--- Version : 1.0 - Beta                                                ---
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity xor_4x8 is
port(x : in std_logic_vector(7 downto 0);
y : in std_logic_vector(7 downto 0);
z : in std_logic_vector(7 downto 0);
w : in std_logic_vector(7 downto 0);
result : out std_logic_vector(7 downto 0));
END xor_4x8;


architecture behavior of xor_4x8 is
component xor_4x1
      PORT(x,y,z,w : in std_logic;
           result : out std_logic);
END component;

begin
aa: for i in 0 to 7 generate
      bb: xor_4x1 port map(x(i),y(i),z(i),w(i),result(i));
    end generate;
end behavior;
```

41. XOR_8X1

```
--------------------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                                 ---
--- ID :  1669                                                         ---
--- Component : This component XORs 8 single bit values.              ---
--- Version : 1.0 - Beta                                               ---
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity xor_8x1 is
port(a,b,c,d,e,f,g,h : in std_logic;
     result : out std_logic);
end xor_8x1;

architecture dataflow of xor_8x1 is
begin
      process(a,b,c,d,e,f,g,h)
      begin
            result <= (a xor b xor c xor d xor e xor f xor g xor h);
      end process;
end dataflow;
```

42. XOR_8X8

```
--------------------------------------------------------------------------
--- Author : Ananda Raja A/L Dore Raja                                 ---
--- ID :  1669                                                         ---
--- Component : This component XORs 8 -> 8 bit values.                ---
--- Version : 1.0 - Beta                                               ---
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity xor_8x8 is
port (a : in std_logic_vector(7 downto 0);
b : in std_logic_vector(7 downto 0);
c : in std_logic_vector(7 downto 0);
d : in std_logic_vector(7 downto 0);
e : in std_logic_vector(7 downto 0);
f : in std_logic_vector(7 downto 0);
g : in std_logic_vector(7 downto 0);
h : in std_logic_vector(7 downto 0);
result : out std_logic_vector(7 downto 0));
end xor_8x8;

architecture behavior of xor_8x8 is
component xor_8x1
      PORT(A,B,C,D,E,F,G,H : in std_logic;
           result : out std_logic);
END component;

begin
aa: for i in 0 to 7 generate
      bb: xor_8x1 port
map(a(i),b(i),c(i),d(i),e(i),f(i),g(i),h(i),result(i));
    end generate;
end behavior;
```